



Bohemia Automation

Cross-platform desktop apps in Rust and Qt



Serhij Symonenko / Bohemia Automation



Desktop vs Web in 2023

- No HTTP: industrial devices, databases, clients for 3rd party web services etc.
- Sometimes desktop apps are just faster than web ones, require less RAM
- Sometimes apps require more access to the local machine than WASM can provide
- No limitations - use any crates, use multiple threads, use async runtimes (tokio)
- Create apps with no HTML and CSS coding
- Users must download and install your app – take in account rules of good taste in your sector





Bohemia Automation

Why Qt?

- Two big grown-up players: Qt and GTK
- Alternatives – Flutter, Slint etc
are either too young or less focused on desktop apps
- Qt usually looks neater and more Enterprise-ready
(but take into account the license)
- Qt perfectly works with 4K screens
- Qt correctly works with resizable windows
- Qt has got commercial support, including lots
of 3rd party consulting companies
- Qt has got Qt Designer and QML
- Qt 5 or 6? Check KDE: <https://iskdeusingqt6.org>





Bohemia Automation

Qt in Rust

- **Ritual**

<https://rust-qt.github.io>

- **qmetaobject**

<https://github.com/woboq/qmetaobject-rs>

- **cxx-qt**

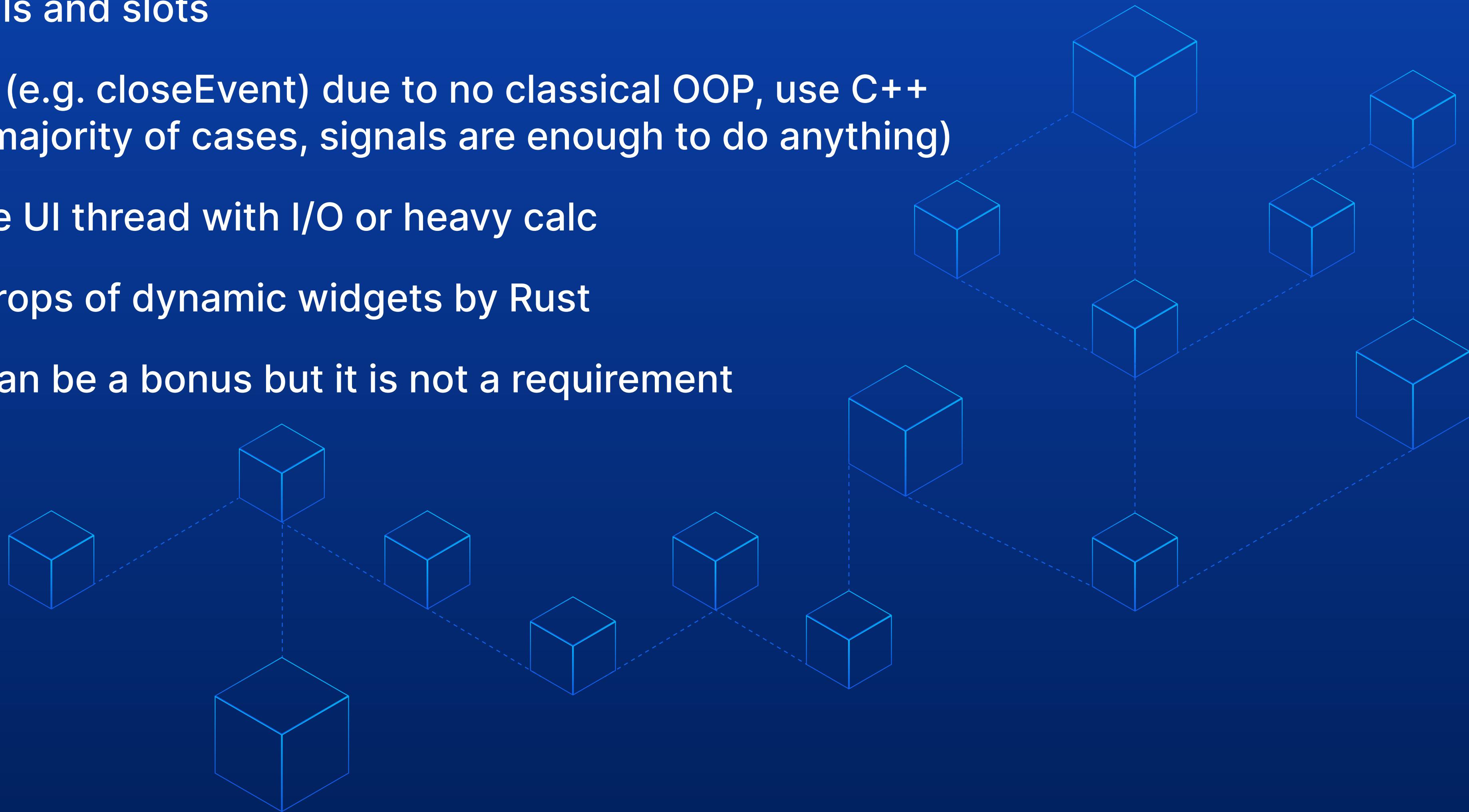
<https://www.kdab.com/cxx-qt/>





Qt basics for rustaceans

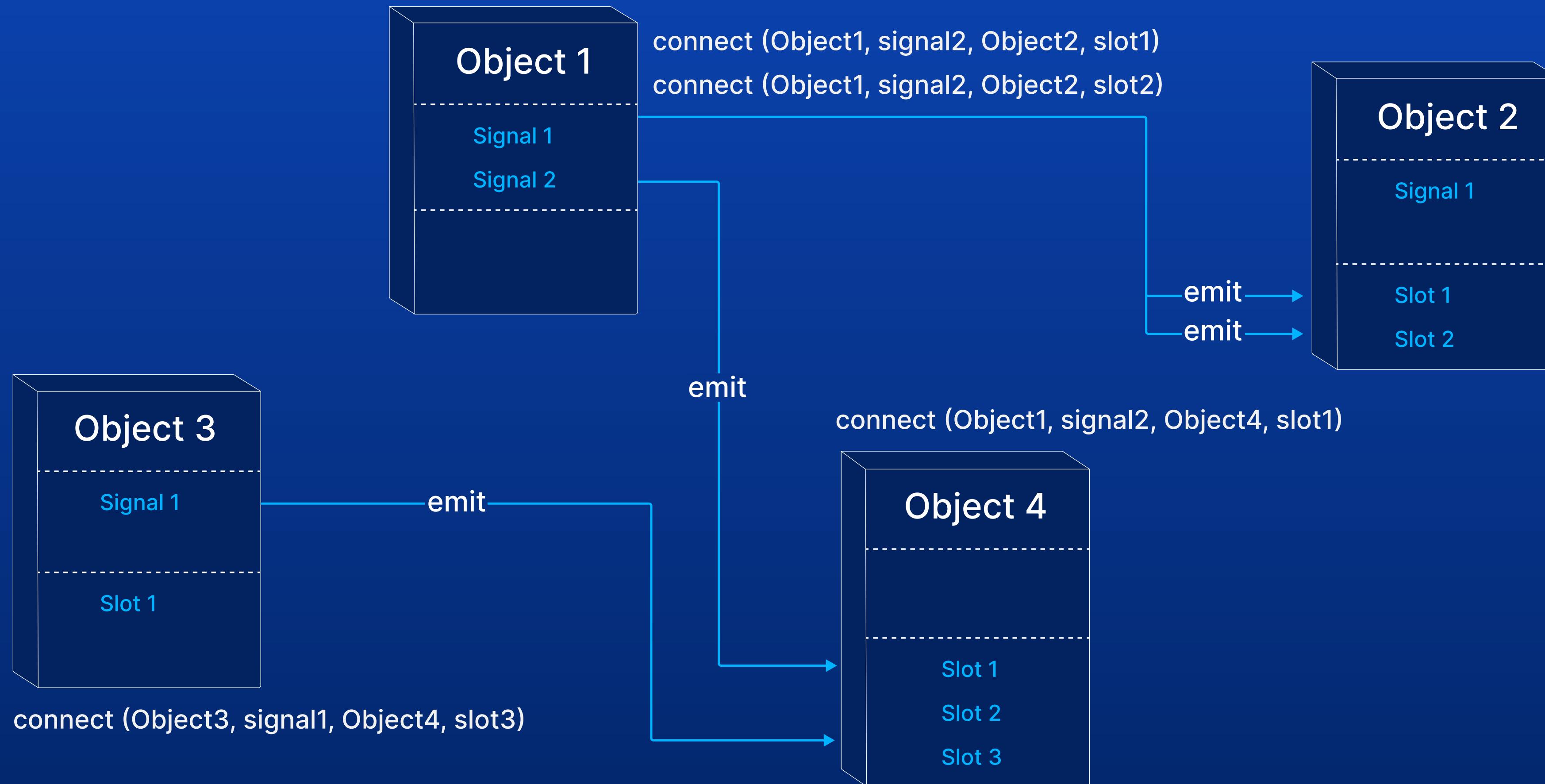
- Learn about signals and slots
- No events in Rust (e.g. `closeEvent`) due to no classical OOP, use C++ wrappers (in the majority of cases, signals are enough to do anything)
- Avoid blocking the UI thread with I/O or heavy calc
- Beware of auto-drops of dynamic widgets by Rust
- C++ knowledge can be a bonus but it is not a requirement





Bohemia Automation

Signals and Slots ecosystem

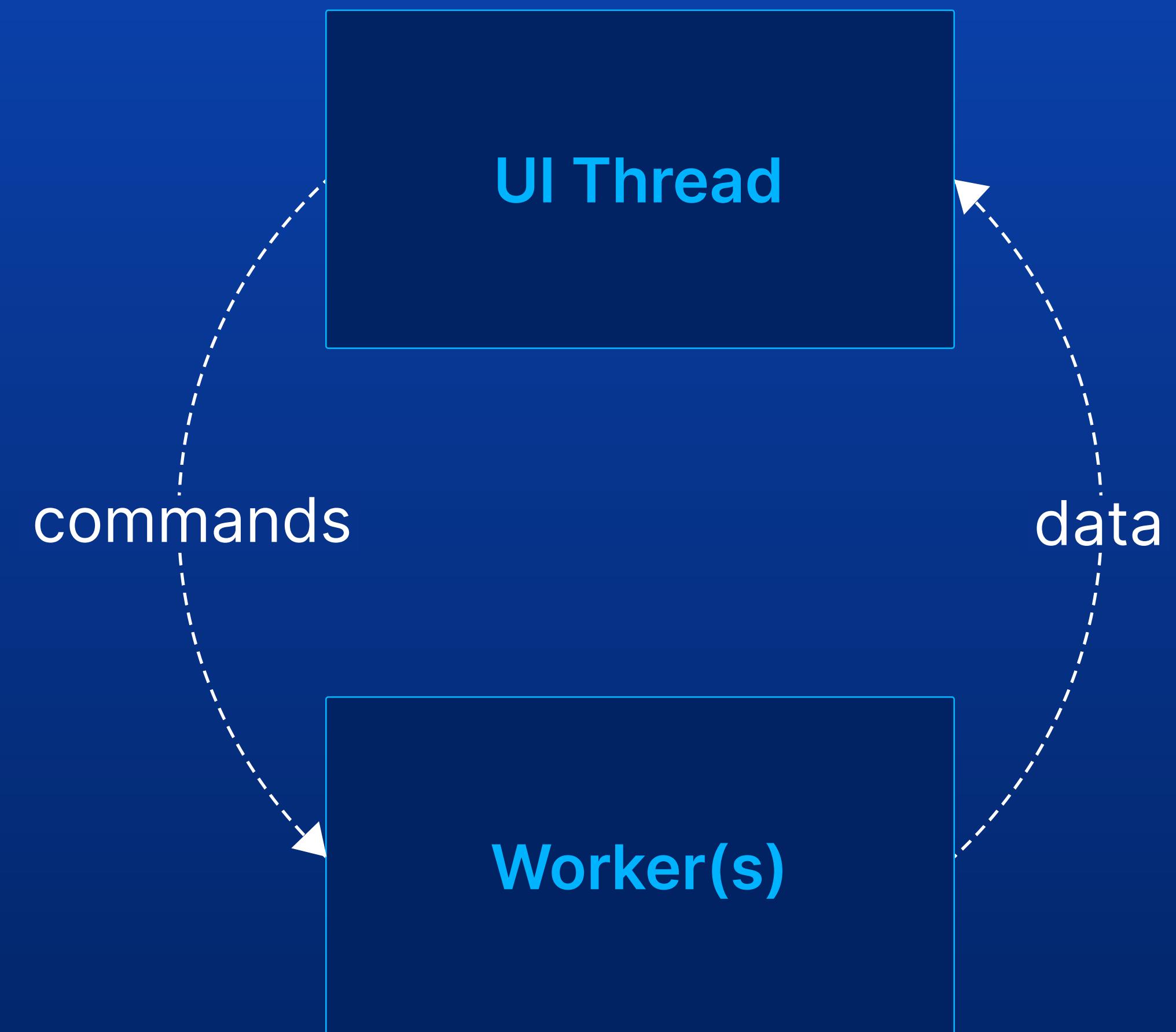


Note: Signal objects are not completely thread-safe but `signal.emit()` function is an exception



Bohemia Automation

Organizing data flow





Why do people still use Ritual

- The oldest and the most popular Rust-Qt project with no serious issues
- The most complete bindings for Qt5
- Supports Qt Designer-generated forms
- Supports dynamic widget creation
- Minimal Rust overhead
 - Qt has got a good documentation and issue discussions. Majority of that can be used with Ritual almost as-is
 - In case of project death and zero alternatives, UI logic can be ported to pure C++ with low costs

When Ritual should be avoided

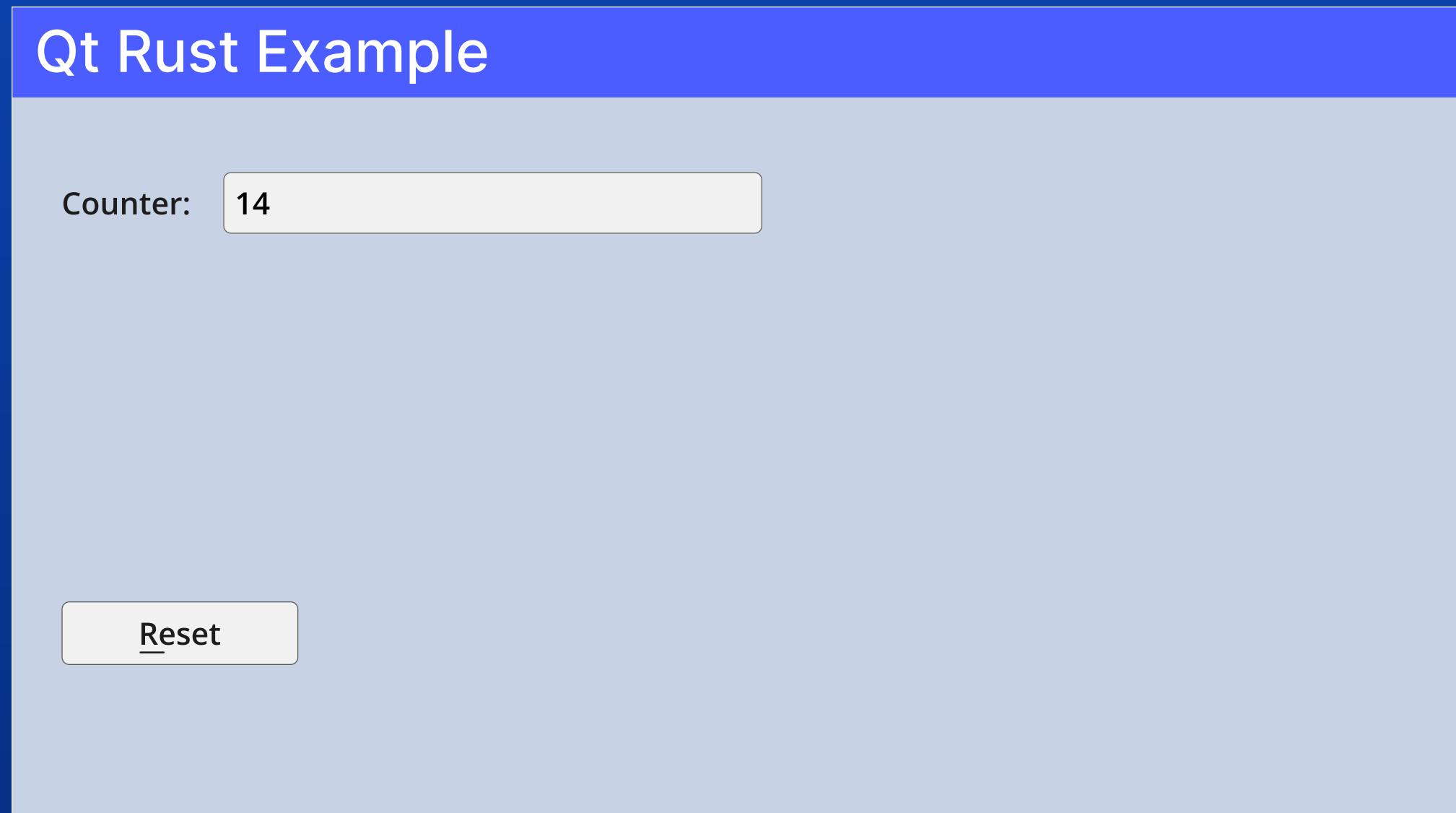
- You prefer to keep away from unsafe blocks in your code
- You use QML only
- You do not care about Qt knowledgebase and prefer a Rust way
- You need Qt6 support





Bohemia Automation

Program example (Ritual)



<https://github.com/divi255/qtx>



- A background worker increases the counter value every second
- When the value is increased, a data event is being sent to the UI thread
- UI reset button can send reset commands to the background worker to reset the counter
- UI and the worker thread talk via standard mpsc channels
- The worker thread uses Qt Signal with no arguments to notify UI thread about new events in the data channel



The main function (Ritual)

Initializing channels

```
1 fn main() {
2     QApplication::init(|_| {
3         let (command_tx, command_rx) = std::sync::mpsc::sync_channel::<Command>(64);
4         let (data_tx, data_rx) = std::sync::mpsc::sync_channel::<Data>(64);
5         let data_signal = UnsafeSend::new(unsafe { SignalNoArgs::new() });
6         // construct UI
7         let ui = Ui::new(command_tx.clone(), data_rx);
8         // connect data signal with UI handle_data slot method
9         unsafe {
10             data_signal.connect(&ui.slot_handle_data());
11             // display the UI (non-blocking)
12             ui.window.widget.show();
13         }
14         // spawn the background worker
15         std::thread::spawn(move || {
16             worker(command_rx, data_tx, data_signal);
17         })
18         // execute the Qt application until user exits (blocking)
19         let result: i32 = unsafe { QApplication::exec() };
20         // optionally terminate the background worker
21         command_tx.send(Command::Quit).unwrap();
22         result
23     })
24 }
```



<https://github.com/divi255/qtx>



The main function (Ritual)

Initializing UI

```
1 fn main() {
2     QApplication::init(|_| {
3         let (command_tx, command_rx) = std::sync::mpsc::sync_channel::<Command>(64);
4         let (data_tx, data_rx) = std::sync::mpsc::sync_channel::<Data>(64);
5         let data_signal = UnsafeSend::new(unsafe { SignalNoArgs::new() });
6         // construct UI
7         let ui = Ui::new(command_tx.clone(), data_rx);
8         // connect data signal with UI handle_data slot method
9         unsafe {
10             data_signal.connect(&ui.slot_handle_data());
11             // display the UI (non-blocking)
12             ui.window.widget.show();
13         }
14         // spawn the background worker
15         std::thread::spawn(move || {
16             worker(command_rx, data_tx, data_signal);
17         });
18         // execute the Qt application until user exits (blocking)
19         let result: i32 = unsafe { QApplication::exec() };
20         // optionally terminate the background worker
21         command_tx.send(Command::Quit).unwrap();
22         result
23     })
24 }
```



<https://github.com/divi255/qtx>



The main function (Ritual)

Spawning Rust workers

```
1 fn main() {
2     QApplication::init(|_| {
3         let (command_tx, command_rx) = std::sync::mpsc::sync_channel::<Command>(64);
4         let (data_tx, data_rx) = std::sync::mpsc::sync_channel::<Data>(64);
5         let data_signal = UnsafeSend::new(unsafe { SignalNoArgs::new() });
6         // construct UI
7         let ui = Ui::new(command_tx.clone(), data_rx);
8         // connect data signal with UI handle_data slot method
9         unsafe {
10             data_signal.connect(&ui.slot_handle_data());
11             // display the UI (non-blocking)
12             ui.window.widget.show();
13         }
14         // spawn the background worker
15         std::thread::spawn(move || {
16             worker(command_rx, data_tx, data_signal);
17         });
18         // execute the Qt application until user exits (blocking)
19         let result: i32 = unsafe { QApplication::exec() };
20         // optionally terminate the background worker
21         command_tx.send(Command::Quit).unwrap();
22         result
23     })
24 }
```



<https://github.com/divi255/qtx>



The main function (Ritual)

Running the application

```
1 fn main() {
2     QApplication::init(|_| {
3         let (command_tx, command_rx) = std::sync::mpsc::sync_channel::<Command>(64);
4         let (data_tx, data_rx) = std::sync::mpsc::sync_channel::<Data>(64);
5         let data_signal = UnsafeSend::new(unsafe { SignalNoArgs::new() });
6         // construct UI
7         let ui = Ui::new(command_tx.clone(), data_rx);
8         // connect data signal with UI handle_data slot method
9         unsafe {
10             data_signal.connect(&ui.slot_handle_data());
11             // display the UI (non-blocking)
12             ui.window.widget.show();
13         }
14         // spawn the background worker
15         std::thread::spawn(move || {
16             worker(command_rx, data_tx, data_signal);
17         });
18         // execute the Qt application until user exits (blocking)
19         let result: i32 = unsafe { QApplication::exec() };
20         // optionally terminate the background worker
21         command_tx.send(Command::Quit).unwrap();
22         result
23     })
24 }
```



<https://github.com/divi255/qtx>



The main function (Ritual)

Shutting down

```
1 fn main() {
2     QApplication::init(|_|
3         let (command_tx, command_rx) = std::sync::mpsc::sync_channel::<Command>(64);
4         let (data_tx, data_rx) = std::sync::mpsc::sync_channel::<Data>(64);
5         let data_signal = UnsafeSend::new(unsafe { SignalNoArgs::new() });
6         // construct UI
7         let ui = Ui::new(command_tx.clone(), data_rx);
8         // connect data signal with UI handle_data slot method
9         unsafe {
10             data_signal.connect(&ui.slot_handle_data());
11             // display the UI (non-blocking)
12             ui.window.widget.show();
13         }
14         // spawn the background worker
15         std::thread::spawn(move || {
16             worker(command_rx, data_tx, data_signal);
17         });
18         // execute the Qt application until user exits (blocking)
19         let result: i32 = unsafe { QApplication::exec() };
20         // optionally terminate the background worker
21         command_tx.send(Command::Quit).unwrap();
22         result
23     })
24 }
```



<https://github.com/divi255/qtx>



UI and Main window

```
1 // main window
2 #[ui_form("../ui/main.ui")]
3 struct Main {
4     letwidget: QBox<QWidget>,
5     counter: QPtr<QLineEdit>,
6     btn_reset: QPtr<QPushButton>,
7 }
8
9 // UI
10 struct Ui {
11     window: Main,
12     command_tx: mpsc::SyncSender<Command>,
13     command_rx: mpsc::SyncReceiver<Command>,
14     data_rx: mpsc::Receiver<Data>,
15 }
16 // required to transform Rust functions into slots
17 impl cpp_core::StaticUpcast<QObject> for Ui {
18     unsafe fn static_upcast(ptr: Ptr<Self>) → Ptr<QObject> {
19         ptr.window.widget.as_ptr().static_upcast()
20     }
21 }
```



<https://github.com/divi255/qtx>



UI Implementation

```
1 impl Ui {  
2     // Rc required to transform Rust functions into slots  
3     fn new(  
4         command_tx: std::sync::mpsc::SyncSender<Command>,  
5         data_rx: std::sync::mpsc::Receiver<Data>,  
6     ) → Rc<Self> {  
7         let ui = Rc::new(Ui {  
8             window: Main::load(),  
9             command_tx,  
10            data_rx,  
11        });  
12        unsafe {  
13            ui.window  
14                .btn_reset  
15                .clicked()  
16                .connect(&ui.slot_handle_btn_reset());  
17        }  
18        ui  
19    }  
20    #[slot(SlotNoArgs)]  
21    fn handle_btn_reset(self: &Rc<Self>) {  
22        let _ = self.command_tx.send(Command::Reset);  
23    }  
24 }
```



<https://github.com/divi255/qtx>



UI Implementation

An alternative slot implementation

```
1 fn new(  
2     command_tx: std::sync::mpsc::SyncSender<Command>,  
3     data_rx: std::sync::mpsc::Receiver<Data>,  
4 ) -> Rc<Self> {  
5     let ui = Rc::new(Ui {  
6         window: Main::load(),  
7         command_tx: command_tx.clone(),  
8         data_rx,  
9     });  
10    // define a slot manually using a code closure  
11    unsafe {  
12        ui.window  
13            .btn_reset  
14            .clicked()  
15            .connect(&SlotNoArgs::new(&ui.window.widget, move || {  
16                // the main object is under Rc as it needs to be cloned to be moved into a slot  
17                // closure if required  
18                let _ = command_tx.send(Command::Reset);  
19            }));  
20    }  
21    ui  
22 }
```



<https://github.com/divi255/qtx>



Worker and Events

```
1 // commands to the background worker
2 enum Command {
3     Reset,
4     Quit,
5 }
6 // data from the background worker (alternative: slots, but objects must be Qt-ized)
7 enum Data {
8     Counter(u64),
9 }
10 // background worker
11 fn worker (
12     command_rx: std::sync::mpsc::Receiver<Command>,
13     data_tx: std::sync::mpsc::SyncSender<Data>,
14     data_signal: UnsafeSend<QBox<SignalNoArgs>>,
15 ) {
16     let mut counter = 0;
17     loop {
18         while let Ok(command) = command_rx.try_recv() {
19             match command {
20                 Command::Reset => counter = 0,
21                 Command::Quit => return,
22             }
23         }
24         if data_tx.send(Data::Counter(counter)).is_ok() {
25             println!("emitting signal from {:?}", std::thread::current().id())
26             unsafe {
27                 data_signal.emit();
28             }
29         }
30         std::thread::sleep(Duration::from_secs(1));
31         counter += 1;
32     }
33 }
```



<https://github.com/divi255/qtx>



Handling data events in UI

```
1 impl Ui {  
2     // .....  
3     // .....  
4     // add the following method to UI  
5     #[slot(SlotNoArgs)]  
6     fn handle_data(self: &Rc<Self>) {  
7         println!("running handle data in {:?}", std::thread::current().id());  
8         while let Ok(data) = self.data_rx.try_recv() {  
9             match data {  
10                 Data::Counter(v) => {  
11                     unsafe {  
12                         self.window.counter.set_text(&qt_core::qs(v.to_string()));  
13                     };  
14                 } // match other data kinds if exist  
15             }  
16         }  
17     }  
18 }
```



<https://github.com/divi255/qtx>



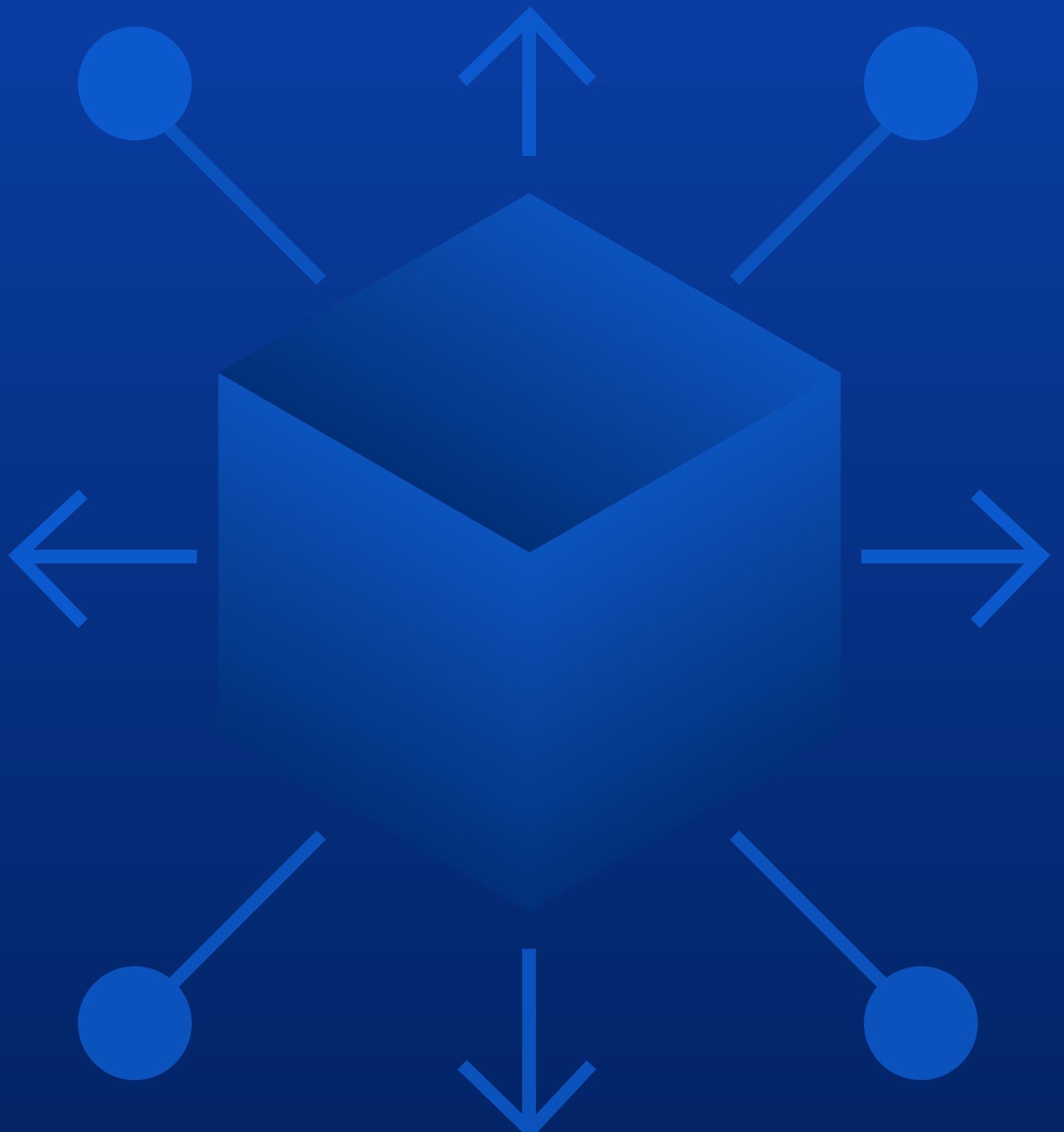
How to distribute apps

- cargo bundle (Linux/OSX)

<https://github.com/burtonageo/cargo-bundle>

- cargo wix (Windows)

<https://github.com/volks73/cargo-wix>





Bohemia Automation

Thank you for watching!

The presentation and the source code:

<https://github.com/divi255/qtx>

Production app example:

<https://info.bma.ai/en/actual/eva4/ecmui/>