For a brighter QFuture

QtCon / Akademy, Berlin 2016

Ivan Čukić

ivan.cukic@kde.org
http://cukic.co
About me

- KDE development
- Talks and teaching
- Functional programming enthusiast, but not a purist
Disclaimer

Make your code readable. Pretend the next person who looks at your code is a psychopath and they know where you live.

Philip Wadler
Disclaimer

The code snippets are optimized for presentation, it is not production-ready code.

std namespace is omitted, value arguments used instead of const-refs or forwarding refs, etc.
FUTURES

Introduction
Concurrency
Futures
T value = function();
What if the call takes too long to complete?

T value = function();

execution is blocked until function finishes
Blocking

- I/O
- User input
- Network communication

```javascript
T value = function();
```

*execution is blocked until function finishes*
Callbacks?
Signals and slots?
Spin off threads, and wait in the thread?

```javascript
T value = function();
```

execution is blocked until function finishes
Inversion of Control
“Spaghetti code”
by George W. Hart
future<T> handler = function();
future<T> handler = function();
...
T value = handler.get();
future<T> handler = function();
...
T value = handler.get();!
Future

```javascript
future<T> handler = function();
...
handler → [] (T value) { ... }
```
future<T> handler = function();

handler.then([] (T value) { ... });

auto value = await handler();
Concurrencey problems

A large fraction of the flaws in software development are due to programmers not fully understanding all the possible states their code may execute in. In a multithreaded environment, the lack of understanding and the resulting problems are greatly amplified, almost to the point of panic if you are paying attention.

John Carmack
In-depth: Functional programming in C++
Futures

std::future
boost::future
QFuture
Folly Future
QFUTURE

History
Problems
Under the wraps
A bit of history

Qt4

Born in **QtConcurrent** – for collecting the results of asynchronous operations

Operations like filtering, mapping, reduction etc. or for simply executing a function on another thread.

The main use-case was the fork-join pattern – do stuff in multiple threads, and get the result.
A bit of history

Road to Qt5

Moving QFuture from QtConcurrent to QtCore

When QtConcurrent was been moved out of QtCore, some of it stayed behind in QtCore: QThreadPool, but not QFuture. I’m arguing here that QFuture should stay in QtCore, or else be renamed to QtConcurrent::Future, to not impede development in that area until Qt 6.

[...]

– Mark Mutz
Qt5

**QFuture** is a part of QtCore and is no more tied to QtConcurrent.

But is it really?

- Still meant only for multi-threading
- It can not really be constructed outside of QtConcurrent
- API tailored exactly for QtConcurrent uses
  ...

Threading

Qt5

Used to model thread-based concurrent concurrent invocations. What about all other asynchronous computations?

- `QMetaObject::invokeMethod` and `QueuedInvocation"If the invocation is asynchronous, the return value cannot be evaluated";`
- `QDBusPendingReply<T>` is a thing. It is a value that will be available in the future;
- `QNetworkReply` is a complex structure which will initialize its data some time in the future;
- `KJob` is again a process that can yield a result when the asynchronous job is completed;
Many future-like things, none of them are \texttt{QFuture}.

No ability to compose several calls. Require a lot of boiler-plate to deal with all of them.

\begin{verbatim}
void processResult(QFuture<Smth> future)
    how awesome would it be not to care about which future-like object it is
\end{verbatim}
So, how do I **create** a QFuture?

It has only the default constructor which creates an empty, canceled future, and a copy constructor.

```cpp
QFuture();
QFuture(const QFuture &other);
```
Construction

Qt5

So, how do I create a QFuture?

From the docs: "To start a computation, use one of the APIs in the Qt Concurrent framework."

```cpp
QtConcurrent::run(...);
QtConcurrent::filter(...);
QtConcurrent::mappedReduced(...);
```
Getting the value

Qt5

So, how do I **get the value** from a QFuture?

We can use `.get()`, but then there is no point in using the futures in the first place.

```java
future.get();
```
Getting the value

Qt5

So, how do I get the value from a QFuture?

Instead of trying to get the value, consider the future is a black box, and we can only tell it to whom to send the result.

\[
\text{future} \rightarrow \text{continuation}
\]
Getting the value

Qt5

So, how do I get the value from a QFuture?

Instead of trying to get the value, consider the future is a black box, and we can only tell it to whom to send the result.

```cpp
auto watcher = new QFutureWatcher<int>();
QObject::connect(watcher,
    &QFutureWatcherBase::finished, [=] {
        continuation(watcher->result());
        watcher->deleteLater();
    });
watcher->setFuture(qfuture);
```
Other fun things

- Not only one value – QFuture<T> is essentially a future of a list of Ts
- It can store an exception (an error in the asynchronous computation), but the exception can not be accessed via the API without calling .get() which rethrows the exception.
- Job control – setPaused(bool), cancel()
Under the wraps

QFuture is very limited as far as the public API is concerned, and we (will pretend) we can not access the private API of a class template.

But, the interesting things are not in the QFuture, but in the QFutureInterface<T>.

```cpp
template <typename T>
class QFutureInterface: ... {
public:
    QFuture<T> future();
};
```
So, how do I create a QFuture?

Creating a future that already holds a value is trivial, just create the interface instance, and set the value.

```cpp
QFutureInterface<T> interface;
auto future = interface.future();

interface.reportStarted();
interface.reportResult(value);
interface.reportFinished();
return future;
```
So, how do I create a QFuture?

To create a future that contains an error – just create the interface instance, and set the error.

```cpp
QFutureInterface<T> interface;
auto future = interface.future();
interface.reportStarted();
interface.reportException(exception);
interface.reportFinished();
return future;
```
So, how do I create a QFuture?

To create a future that will contain a value \textit{after a few seconds} you’ll have to do a bit more. Create your own future interface class.

```cpp
template<typename T>
class DelayedFutureInterface : public QObject,
    public QFutureInterface<T>
{
    ...
};
```
And make its start member function complete the future after a given number of milliseconds.

```cpp
QFuture<T> start()
{
    auto future = this->future();
    this->reportStarted();

    QTimer::singleShot(milliseconds, [this] {
        this->reportResult(value);
        this->reportFinished();
        deleteLater();
    });

    return future;
}
```
AsynQt

So, how do I create a QFuture?

makeReadyFuture(6);
makeCanceledFuture<\texttt{void}>();
makeDelayedFuture(42, 1\texttt{h} + 30\texttt{min}); // \texttt{C++14}
Construction

**AsynQt**

So, how do I create a QFuture?

Wrappers can be written once, and then everything becomes a QFuture.

```cpp
DBus::asyncCall(...);

Process::getOutput("ls");

Process::exec("ls", [](auto p) { return p->exitCode(); });
```

collects process output

the future will contain the process exit code
Getting the value

So, how do I **get the value** from a QFuture?

You simply **don’t**.

But you can pass the value on, once it is available.

```cpp
QFuture<QString> input = getUserInput();

we can not create a .then for the QFuture
input | [] (QString) { do something with the value };

also returns a future
```
AsynQt

QFuture<QString> input = getUserInput();

QFuture<int> length = input | transform(&QString::length);

future that will be initialized as soon as the input becomes available

QFuture<QString> valid = input | filter(&inputValidation);

future that will hold only valid input strings (remember, QFuture can hold a list of items)
Transforming the value

```cpp
QFuture<QByteArray> future = Process::getOutput("echo", { "Hello KDE" });

we got a future of QByteArray
but we wanted QFuture<QString>

QFuture<QString> castFuture = qfuture_cast<QString>(future);

QFuture<QString> castFuture = future | cast<QString>();

or just simply pipe it
```
Getting the value

AsynQt

What if we want to send the value to another function that will return us a QFuture, will we get a `QFuture<QFuture<T>>`?

```cpp
QFuture<QString> input = getUserInput();

QFuture<int> length =
    creates a nested future
    flatten(input | transform( ... ));

QFuture<int> length =
    converts a nested future into a normal one
    input | [] (QString value) {
        shorthand for transform-and-flatten
        // server returns us a future of the HTTP status
        return server.send(value);
    };
```
More composing

**AsynQt**

And the usual...

**collect**: collection\<QFuture\<T\>> \(\rightarrow\) QFuture\<collection\<T\>>

**collect**:

\((\text{QFuture}\<T_1\>, \text{QFuture}\<T_2\>, \ldots)\)

\(\rightarrow\) QFuture\<tuple\<T_1, T_2, \ldots\>>

**any0f**: collection\<QFuture\<T\>> \(\rightarrow\) QFuture\<T\>

**any0f**:

\((\text{QFuture}\<T_1\>, \text{QFuture}\<T_2\>, \ldots)\)

\(\rightarrow\) QFuture\<variant\<T_1, T_2, \ldots\>>

...
Limitations of QFuture

- Either single values, or multiple values stored in memory – not suitable for data streams;
- Custom QFutures are not first-class citizens;
- Big overhead (both API and runtime) for a concept that could have been much simpler if only it weren’t born as a part of Qt Concurrent;
- AsynQt – useful or a fun, but futile experiment?
Answers? Questions! Questions? Answers!

Kudos:

Friends at KDE
Saša Malkov
Zoltán Porkolab

Worth reading and watching:

- Systematic Error Handling in C++, Andrei Alexandrescu
- Await 2.o, Gor Nishanov
- Ranges proposal, Eric Niebler