How to return errors from APIs

How to return errors from APIs

For the most part, we should follow the patterns of Qt's APIs. (Unfortunately Qt hasn't documented their error strategies explicitly. So we attempt to here.)

see also: How to Log Errors

What *is* an Error

> ⚠ An *Error* occurs when a called function could not complete its *primary* purpose.

see also: Error Handling Thoughts

Table of Contents

Returning Synchronous Errors

Call a function. It fails. How to indicate the error to the calling code.
Note that these points are accumulative...

0. Whenever possible, primary values (the purpose of the function) should be returned directly (not as out params)

BAD

```
void dateOfBirth(QDateTime * date);
void favouriteNumber(int * number);
void square_root(double val, double *
root);
void fireMissile(Direction direction);

enum BatteryState
{
   Charging,
   FullyCharged,
   ...
};

void batteryState(BatteryState *
state);
```

OK

```
QDateTime dateOfBirth();
int favouriteNumber();
double square_root(double val);
void fireMissile(Direction
direction);

enum BatteryState
{
   Charging,
   FullyCharged,
   ...
};

BatteryState batteryState();
```

1. If possible, encode the error as one of the return values

OK

```
QDateTime dateOfBirth();
int favouriteNumber();
double square_root(double
val);
void
fireMissile(Direction
direction);

enum BatteryState
{
   Charging,
   FullyCharged,
   ...
};

BatteryState
batteryState();
```

Good

```
QDateTime dateOfBirth(); // check
QDateTime::isValid()
int favouriteNumber();  // returns -17 on error,
as everyone hates -17
double square_root(double val); // returns -1 on
error
Error fireMissile(Direction direction);

enum BatteryState
{
   Unknown,
   AccessDenied,
   Charging,
   FullyCharged,
   ...
};

BatteryState batteryState();
```

Notes:

- if the primary purpose is an action, and not retrieval of a value, then the error/result is the return type

2. For extra error info, or if all possible return values are valid, use an out-param for the error case

Good

```
QDateTime dateOfBirth(); // check
QDateTime::isValid()
int favouriteNumber();  //
returns -17 on error, as everyone
hates -17
double square_root(double val);
// returns -1 on error
Error fireMissile(Direction
direction);

enum BatteryState
{
   Unknown,
   AccessDenied,
   Charging,
   FullyCharged,
   ...
};

BatteryState batteryState();
```

Better

```
QDateTime dateOfBirth(Error * error); //
check *error for the reason for the
error
int favouriteNumber(bool * ok);  // all
numbers valid, *ok set to false on error
double square_root(double val); //
returns -1 when val is negative
Error fireMissile(Direction direction);

enum BatteryState
{
   Unknown,
   AccessDenied,
   Charging,
   FullyCharged,
   ...
};

BatteryState batteryState();
```

Note:

- the function still needs to be able to return a value even in the error case. Typically a default or invalid value. ie 0 or -1 or an invalid `QDateTime` etc. Most Qt classes have an invalid state (typically the default constructed state) and an `isValid()` function to check the state.

3. Allow the error to be ignored

Somewhat unfortunately, we allow the caller to ignore an error by passing `NULL` for the error out-param. Assume they will either check the validity of the return value (ie `QDateTime::isValid()`, or they are OK with using the error/default value (ie using 0 as the favouriteNumber whether it was an error or not).

Better

```
QDateTime dateOfBirth(Error *
error); // check *error for the
reason for the error
int favouriteNumber(bool * ok);  //
all numbers valid, *ok set to false
on error
double square_root(double val); //
returns -1 when val is negative
Error fireMissile(Direction
direction);

enum BatteryState
{
   Unknown,
   AccessDenied,
   Charging,
   FullyCharged,
   ...
};

BatteryState batteryState();
```

Best

```
QDateTime dateOfBirth(Error * error =
0); // check *error for the reason for
the error
int favouriteNumber(bool * ok = 0);
// all numbers valid, *ok set to false
on error
double square_root(double val); //
returns -1 when val is negative
Error fireMissile(Direction
direction);

enum BatteryState
{
   Unknown,
   AccessDenied,
   Charging,
   FullyCharged,
   ...
};

BatteryState batteryState();
```

4. Document the function and the error cases clearly. Do not leave error cases undefined.

There are actually good reasons to have *preconditions*, and thus undefined behaviour when those preconditions are not met (see discussion at Error Handling Thoughts and How to Log Errors). But for platform APIs it is typically better to avoid undefined behaviour, and instead return to the calling code, even when you suspect the calling code has a bug.

Undefined

```
/*!
 * @brief calculates
approximate square root
 * @param val value of which
to take the square root. MUST
NOT BE NEGATIVE.
 * @returns the positive
square root of the input
value.
*/
double square_root(double
val);
```

Documented

```
/*!
 * @brief calculates approximate square
root
 * @param val value of which to take the
square root. Should not be negative.
 * @returns the positive square root of the
input value. (Returns -1 if the input value
is negative.)
*/
double square_root(double val);
```

. Examples

Best

```
/*!
 * @brief returns user's date of birth
 * @param error if not null, is set to error condition if date of birth could not be
retrieved
 * @return user's date of birth, or an invalid date (isValid() == false) if it could
not be retrieved
*/
QDateTime dateOfBirth(Error * error = 0);

/*!
 * @brief returns user's favourite number
 * @param ok if not null, is set to true if number is retrieved and set to false if
number could not be retrieved.
 * @return user's favourite number, or 0 if favourite number could not be known. As
0 is a valid favourite number, use @c ok param to determine success.
*/
int favouriteNumber(bool * ok = 0);

/*!
 * @brief calculates approximate square root
 * @param val value of which to take the square root. Should not be negative.
 * @returns the positive square root of the input value. (Returns -1 if the input
value is negative.)
*/
double square_root(double val);

/*!
 * @brief fire Missile in the general direction
 * @param direction (and magnitude) of firing. Should be a valid Direction vector.
 * @returns Error::None is successfully fired, else an error code representing the
error.
*/
Error fireMissile(Direction direction);

enum BatteryState
{
   Unknown,
   AccessDenied,
   Charging,
   FullyCharged,
   ...
};

/*!
 * @brief retrieves the current state of the battery
 * @returns the current state of the battery or an error if the state could not be
determined.
*/
BatteryState batteryState();
```

Returning Asynchronous Errors

Much of the functionality of BB10 happens asynchronously, and this is reflected in our APIs. Instead of a single function like

```
int favouriteNumber();
```

we need separate *send* and *receive* mechanisms to send the request for the function to start, then later receive the result of the function. We use normal (synchronous) functions for the send/request, and Qt's signals-slots mechanism to receive the asynchronous results:

```
RequestId requestFavouriteNumber(Error * error = 0);  // send
...
//signal:
void favouriteNumber(RequestId id, int favouriteNumber);  // receive
```

Note:

- whenever the context isn't obvious, some 'key' (like a RequestId) is needed to associate the signal with the corresponding request
- making the request (first function) could fail. Handle this with the same rules (above) as a normal function call. ie, in this case `RequestId` of `0` could mean failure, with `error` holding additional info.
- assuming the request is issued successfully, there could still be a failure later, during async processing. The following points elucidate how that should be done...

Note that the following rules for async error reporting should mirror the rules for synchronous error reporting...

1. If possible, encode the error as one of the signal values

Error as just-another-value

```
    Error requestBatteryState();

    enum BatteryState
    {
       Unknown,
       AccessDenied,
       Charging,
       FullyCharged,
       ...
    };

Q_SIGNALS:
    void batteryState(BatteryState state);
```

2. For extra error info, or if all possible values are valid, use an extra param for the error case

Error as extra param

```
    RequestError requestDateOfBirth();

Q_SIGNALS:
    void dateOfBirth(QDateTime date, AsyncError error);
```

Note that there are 2 different types of errors - the (synchronous) errors for when making the request fails (`RequestError`), and the async error

of processing the request ((`AsyncError`). You should pick better names than this, but it is important that these are different error enums - typically the set of possible errors for each case are separate. (If only one enum was used for both the request and the signal (say `DateOfBirthError`) then the code making the request, and the code responding to the signal, would each likely have `case` statements for handling errors that could never happen at that point.)

2.5 This answers the "one signal or two" question

Note that the above answers the "one signal or two" question (ie should the async error be a *separate* signal, such as `dateOfBirthFailed(AsyncError error)`). See also Error Handling Thoughts and http://overflow.rim.net/questions/30596/one-signal-or-two-for-a-status-signal.

To be explicit: the answer is *one signal*.

This isn't perfect, and there are reasons on both sides, however, with one signal we have:

- sync error rules and async error rules are somewhat parallel
- no one will forget to connect to the second (error) signal
- it is consistent with signals where the error is part of the primary value (ie BatteryState::AccessDenied, BatteryState::Charging,...)
- there is no confusion about whether a 'finished()' signal is still sent after an 'error()' signal
- there is no confusion about having 3 signals (success(), error(), then finished())

And I won't list here the arguments for the other side. 

3. Allow the error to be ignored

Given the above `dateOfBirth` signal, a developer can choose to ignore the async error via Qt's ability to drop params across connections:

Extra param ignored

```
    connect(userInfo, Q_SIGNAL(dateOfBirth(QDateTime, AsyncError)), this,
Q_SLOT(onDateOfBirth(QDateTime)));
```

Note that the signal sends two params - `(QDateTime, AsyncError)`, but the slot only receives the first one `(QdateTime)`.
Again, we don't recommend ignoring error info, but some code behaves the same no matter what the error reason is - it only cares that there was *some* error. Hopefully on`DateOfBirth()` at least checks `date.isValid()` to see that there was some error.

4. Document it!

Be clear that the request function can fail immediately, and that the async signal that comes back could also indicate failure.

Request	Signal Received

`/*!`
` * @brief requests the user's date of birth`
` * @returns RequestResult::Success if the request was successfully sent to the service`
` *  and thus a @c dateOfBirh() signal should be expected,`
` *  otherwise an error code indicating why the request could not be sent`
` *  and thus no signal will be emitted.`
`*/`
`UserInfo::RequestResult requestDateOfBirth();`
	`/*!`
	` * @brief Receives the result of requestDateOfBirth().`
	` * @param birthDate user's date of birth,`
	` *  or an invalid date if the date of birth could not be retrieved`
	` * @param error DateOfBirthError::None on success,`
	` *  otherwise an indication of why the date of birth could not be retrieved.`
	`*/`
	`void dateOfBirth(QDateTime const & birthDate, UserInfo::DateOfBirthError error);`

. Final Examples

Request	Signal Received

`/*!`
` * @brief requests the user's date of birth`
` * @returns RequestResult::Success if the request was successfully sent to the service`
` *  and thus a @c dateOfBirh() signal should be expected,`
` *  otherwise an error code indicating why the request could not be sent`
` *  and thus no signal will be emitted.`
`*/`
`UserInfo::RequestResult requestDateOfBirth();`
	`/*!`
	` * @brief Receives the result of requestDateOfBirth().`
	` * @param birthDate user's date of birth,`
	` *  or an invalid date if the date of birth could not be retrieved`
	` * @param error DateOfBirthError::None on success,`
	` *  otherwise an indication of why the date of birth could not be retrieved.`
	`*/`
	`void dateOfBirth(QDateTime const & birthDate, UserInfo::DateOfBirthError error);`

```
/*!
 * @brief Starts the calculation of a
square root.
 * @param val value of which to take
the square root. Should not be
negative.
 * @returns RequestResult::Success if
the request was successfully sent to
the service
 *          and thus a @c
squareRoot() signal should be
expected,
 *          otherwise an error code
indicating why the request could not
be sent
 *          (eg
RequestResult::InvalidParam if val is
negative)
 *          and thus no signal will
be emitted.
*/
MathService::RequestResult
requestSquareRoot(double val);
```

```
/*!
 * @brief Receives the result of
requestSquareRoot().
 * @param inputVal  the value from
the original request
 * @param root      the resultant
root, or -1 on error
 * @param error     ResultError::None
on success, otherwise an indication
of the error
*/
void squareRoot(double inputVal,
double root, MathService::ResultError
error);
```

```
/*!
 * @brief fire Missile id in the
general direction.
 * @param id the missile to fire
 * @param direction the direction
(and magnitude) to fire the missile
 * @param code authorization to fire
the missile
 * @returns RequestResult::Success if
the request was successfully sent to
the service
 *          and thus a @c
missileFireAttempted() signal should
be expected,
 *          otherwise an error code
indicating why the request could not
be sent
 *          (eg
RequestResult::AccessDenied)
 *          and thus no signal will
be emitted.
*/
defense::RequestResult
fireMissile(MissileId id, Direction
direction, AccessCode code);
```

```
/*!
 * @brief Emitted when a missile is
fired, or fails to fire. see @c
fireMissile().
 * @details Note that this only
reports on whether the missile was
launched or not.
 *          If launched successfully,
also expect signals like @c
missileUpdated()
 *          emitted periodically
during the missile's journey.
 * @param id the missile in question.
 * @param result result of the
attempt, including
FireResult::Success if successfully
fired,
 *        or an error code if the
missile could not be fired.
*/
void missileFireAttempted(MissileId
id, Defense::FireResult result);
```

```
/*!
 * @brief Sends a request for the state of the battery.
 *        Battery state will be returned via the batteryState() signal.
 * @returns RequestResult::Success if the request was successfully sent to the service
 *          and thus a @c batteryState() signal should be expected,
 *          otherwise an error code indicating why the request could not be sent
 *          (see @c BatteryInfo::RequestResult for possible error codes)
 *          and thus no signal will be emitted.
*/
BatteryInfo::RequestResult requestBatteryState();
```

```
/*!
 * @brief Receives the result of requestBatteryState().
 * @param state  the current state of the battery, or an error if the state could not be retrieved.
*/
void batteryState(BatteryInfo::State state);
```

```
/*!
 * @brief Sets the data item for a
given key from a cloud node.
 * @detail Cloud nodes can hold
arbitrary data items, with each item
associated with a string key.
 *         This function sets the
data for the given key.
 *         Existing data on that key,
if any, is removed and replaced with
the new data.
 *         This is an async task.
Completion is communicated via the
@dataChanged() signal.
 *         Note: a signal *is*
emitted even if old data == new data
(ie even if there was no "change")
 * @param node The node on which the
data will reside.
 * @param key The key that the data
is associated with.
 * @param data The data to be
associated with the key.
 * @returns RequestResult::Success if
the request was successfully sent to
the service
 *          and thus a @c
dataChanged() signal should be
expected,
 *          otherwise an error code
indicating why the request could not
be sent
 *          (eg
RequestResult::AccessDenied)
 *          and thus no signal will
be emitted.
*/
cloud::RequestResult
setData(cloud::Node node, QString
key, QVariant data);
```

```
/*!
 * @brief emitted when the data
associated with key on the node has
changed.
 * @param node the node on which the
change occurred.
 * @param key the key the data is
associated with.
 * @param oldData a unique id for the
old data. The actual data may be
retrieved
 *        (if still available,
depending on the node's data
retention, undo, and versioning
settings) via requestData().
 * @param newData a unique id for the
new data. The data may be retrieved
(at least until overwritten) via
requestData().
*/

void dataChanged(cloud::Node node,
QString key, cloud::DataId oldData,
cloud::DataId newData);
```

```
/*!
 * @brief Performs the given action
on the cloud node.
 * @param node The node on which the
action will be performed.
 * @param action The action to
perform.
 * @param data data that may be
required for certain actions.
 * @param error if not null, set to
RequestError::None on success
 *              and an error code if
the request could not be sent.
 * @returns a unique id representing
the request, otherwise
RequestId::Invalid on error.
 *          This id will be sent as
part of the actionResult() signal.
 *          If the returned id ==
Invalid, no signal is emitted.
*/
cloud::RequestId
performAction(cloud::Node node,
cloud::Action action, QVariant data,
cloud::RequestError * error = 0);
```

```
/*!
 * @brief emitted an action on a node
has been completed.
 * @param node the node on which the
action occurred.
 * @param requestId the id of the
request, returned from
performAction().
 * @param result The result of the
action, if any.
 * @param error ActionError::None if
no error occurred, otherwise a code
indicating the reason for the error.
*/
void actionResult(cloud::Node node,
cloud::RequestId requestId, QVariant
result, cloud::ActionError error);
```

See other guidelines for proper signal, function, and param naming. I also left out a few "`const &`"s for brevity.

Naming: Error vs Result

- Qt tends to use enums like `FooError` and functions like `error()`.
- "Success" should never be used with "Error" ie `Error::Success` is just wrong.
- Use "None" ie `Error::None`.

- The alternative is `FooResult`.
- since the result isn't always an error, this may make more sense.
- Use "Success". ie `FooResult::Success`.

- In the end, either `FooResult` or `FooError` is acceptable.

Reporting Internal Error(s)

You may detect a number of "internal" errors. Like the pps object that the API relies on is missing. Or contains bad data. Or the state of your internal object just doesn't make sense. Likely this means the API cannot accomplish its primary purpose. Thus you need to return an error to the calling code.

What should the error code(s) be?

There is not much the caller can do to change the situation to "fix" internal errors. Most likely, even if you have 10 different internal checks and 10 different reasons for internal errors (pps not found, pps empty, fooPrivate.somePtr == NULL, etc) they will all be handled the same way by the caller - just back out of the current task and tell the user "sorry, couldn't do X".

So there is no sense having multiple different codes for all of your different internal errors. A single `FooResult::InternalError` or `BatteryState::Unknown` is sufficient.

```
enum BatteryState
{
   ...
   Unknown,
   ...
};

enum RequestResult
{
   ...
   InternalError,
   ...
};
```

You should, of course, *log* more information than that (see How to Log Errors), which might help the App developer or the API developer, but you don't need to return any additional information to the calling code.