Qt Engine Edition

December 2016

Based on Qt 5.8

Contents

Contents

Contents

Contents

Networking	TCP/UDP Sockets WebSockets SSL Sockets QNetworkAccessManager Requests and Replies DNS and Proxies Cookies
WebEngine	Qt WebEngine Widgets Handling Asynchronous Functions Exposing Qt objects to JavaScript Engine

Contents

Objectives

Learn…

› …creating and deploying libraries

› …loading libraries and resolving symbols in libraries

› …creating Qt plugins using both high and low-level APIs

Libraries and Plugins

- Application business logic, i.e. an engine can be implemented in a shareable library

- Library
  - A file sharing data and code
  - Can be statically or dynamically linked
    - For static linking, Qt must be configured with `-static` option
  - Loaded at application startup-time
  - Can be loaded and unloaded dynamically

- Plugin
  - A library, implementing an interface
  - Typically several different implementations of the same interface
  - Loaded dynamically, when needed
  - In static builds, plugins may be linked statically, but not loaded in run-time

Custom Libraries

- In general, different platforms handle exporting symbols from a DLL in different ways
  - Some even require a special import declaration when clients of the DLL are compiled
  - Visibility of the symbols of a DLL might also depend on the compiler!
- Once again, Qt hides all this behind a couple of macros:
  - `Q_DECL_EXPORT` – used with symbols *when compiling a shared library*
  - `Q_DECL_IMPORT` – used with symbols *when compiling a client that uses the shared library*
- QtCreator project wizard creates this automatically
- Qt uses private-implementation pattern to guarantee binary compatibility in libraries
  - Public class has a pointer to the private class
  - Private class contains all other data members

Library Deployment

- `DESTDIR` `.pro` file variable defines, where the target file (library) is installed
- Another option is to use `make install` and define files to be installed in `INSTALLS` variable

```
installFiles.files += $${HEADERS}
installFiles.path = $$[QT_INSTALL_HEADERS]
target.path = $$[QT_INSTALL_LIBS]
INSTALLS += target installFiles
```

Library Usage

- The project, using the library, needs to know the location of library headers and binaries
- The easiest way is to put all library-related definitions into either
  - a project include file (`.pri`) or
  - project feature file (`mkspecs/features/*.prf`)
- Use `include(someLibrary.pri)` or `CONFIG += someLibrary.prf` to add definitions to your project

```
# .pro, .pri or .prf file
INCLUDEPATH += $$[QT_INSTALL_HEADERS]
LIBS += -L$$[QT_INSTALL_LIBS]
LIBS += -ldemoLibrary  # No prefix or platform-specific suffix
```

Dynamic Loading and Unloading Libraries

- `QLibrary` allows dynamic explicit library loading/unloading
  - `QLibrary library("simpleLibrary"); // Or use absolute path`
  - Overloaded constructor can be used to give the version number
  - `fileName()` returns the full library name, if the load was successful
- `resolve()` resolves symbols, exported as C functions from the library
  - It also loads the library, if needed
  - `extern "C" SHARED_EXPORT double pow(int a, int b) { }`

```
QLibrary library("simpleLibrary");
typedef double (*PowerFunction)(int, int);
PowerFunction power = (PowerFunction) library.resolve("pow");
if (power)
    qDebug() << power(5, -3);
else
    qDebug() << "Library load failed:" << library.errorString();
```

ex-dynamic-loading

Extending Qt with Plugins

1. Define one or more interfaces

1. Create a plugin project using QtCreator

2. Implement the interfaces – Export the plugin with a JSON file, containing plugin meta data

1. Build and deploy the plugin

1. Load and use the plugin

Low and High-Level Plugin APIs

- Low-level API
  - Allows implementing plugins to extend Qt applications
- High-level API
  - Used to extend Qt itself with plugins
  - Developers need to implement Steps 2-4 only
  - Typically, Step 5 is implemented in plugin factory classes

Step 1: Define One or More Interfaces

- Interface may be a class, containing pure virtual functions only, or it may be an abstract class
  - Classes should not have data members, though
  - Interfaces should derive from `QObject`
  - Interface implementation must derive from `QObject` anyway

- `Q_DECLARE_INTERFACE()` macro tells Qt (meta-object system) about the interface(s)
  - `Q_DECLARE_INTERFACE(CoolInterface, "io.qt.CoolInterface")`
  - The second parameter is an identifier, which is used to register the class, implementing the interface

```
Class CoolInterface
{
public:
    virtual QStringList method1() const = 0;
    virtual QImage method2(const QString &string) = 0;
};
Q_DECLARE_INTERFACE(CoolInterface, "io.qt.CoolInterface")
```

Step 2: Create a Plugin Project Using QtCreator

- The default plugin base class (`QGenericPlugin`) is replaced with the plugin-specific base class
  - Add the plugin-specific function, which creates/registers the plugin object

- If the low-level API is used, subclass `QObject` and your interface
  - Provide any function, e.g. just a constructor, which creates the plugin

Plugin entry point	Created type	Create function
`QPlatformIntegrationPlugin`	`QPlatformIntegration`	`create(const Qstring & key)`
`QStylePlugin`	`QStyle`	`create(const Qstring & key)`
`QQmlExtensionPlugin`	`QQuickItem`	`void registerTypes(const char *uri)`
`QGenericPlugin`	`QObject`	`create(const Qstring & key)`
`Custom`	`Derived from QObject and custom interface`	`NA`

Step 3: Implement the Interfaces

- The plugin class must register the interface it implements
  - Only one interface can be registered in one class
  - Interface identifier (`IID`) must match the identifier, defined in the interface declaration (Step 1)
  - Some plugins require a JSON file, containing meta data about the plugin implementation

- If the low-level API is used, the plugin class must report all interfaces it implements using `Q_INTERFACES` macro

```
Class CoolPlugin : public QObject, public CoolInterface
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID "io.qt.CoolInterface" FILE "coolInterface.json")
    // org.qt-project.Qt.QStyleFactoryInterface IID for QStylePlugin
    // org.qt-project.Qt.QQmlExtensionInterface IID for QQmlExtensionPlugin
    // org.qt-project.Qt.QGenericPluginFactoryInterface IID for QGenericPlugin
    Q_INTERFACES(CoolInterface) // Only needed in low-level API
```

Plugin Meta-Data

- Plugin dependent
- Provides information about the plugin
  - No need to load the plugin library to access this information

Plugin entry point	JSON data

`QImageIOPlugin`	Required, contains supported image formats and MIME types
	`{ "Keys": [ "jpg", "jpeg" ],`
	`"MimeTypes": [ "image/jpeg", "image/jpeg" ] }`

`QStylePlugin`	Required, contains supported style names `{ "Keys": [ "mystyleplugin" ] }`
`QQmlExtensionPlugin`	Not required, plugin info is read from the `qmldir` file
`QGenericPlugin`	Optional, may contain any custom data
`Custom`	Optional, may contain any custom data

Step 4: Build and Deploy the Plugin

- The plugin project file should contain at least the following lines:
  - `TEMPLATE = lib`
  - `CONFIG += plugin`

- The project file should define, where to place the plugin
  - Do so using one of `qmake`'s `DESTDIR` or `INSTALL` variables

```
# Installs target
DESTDIR = $$[QT_INSTALL_PLUGINS]/generic

# Installs any resources when make install executed
target.files += anyFileToBeInstalled
target.path = $$[QT_INSTALL_PLUGINS]/generic
INSTALLS += target
```

Step 5: Load and Use the Plugin
High-Level API Plugins

- Plugins exist in plugin-specific subfolder in `$$[QT_INSTALL_PLUGINS]`
  - E.g. `plugins/styles`
- Additional search paths can be added with
  - `QCoreApplication::addLibraryPath()` or set by `QCoreApplication::setLibraryPaths()`
  - and queried with `QLibraryInfo::location(QLibraryInfo::PluginsPath)`

- Plugins are loaded by factory classes
  - `static QStyle *QStyleFactory::create(const QString &key);`
  - `static QObject *QGenericPluginFactory::create(const QString &key, const QString &specification);`
- Often plugin loading hidden from the developer
  - QML extension plugin loaded by the QML engine
  - `QIOImagePlugin` loaded by `QImageReader`, when `QImage::load(const QString &file)` called

Step 5: Load and Use the Plugin
Low-Level API Plugins

- Plugins must be loaded by the developer with `QPluginLoader`
  - Check, if the plugin is linked against the same Qt version as the loading application

- Similar behavior to `QLibrary` except the plugin object is created with `instance()` function
  - No resolving needed

```
Q_FOREACH(const QString &fileName, pluginsDir.entryList(QDir::Files))
{
    QPluginLoader loader(pluginsDir.absoluteFilePath(fileName));
    QObject* plugin = loader.instance();
    if (plugin) {
        FilterInterface* filter = qobject_cast<FilterInterface*>(plugin);
        if (filter) {…}
    }
    // Plugin unloaded from memory after all QPluginLoader objects of the same
    // library destructed
}
```

Questions and Answers

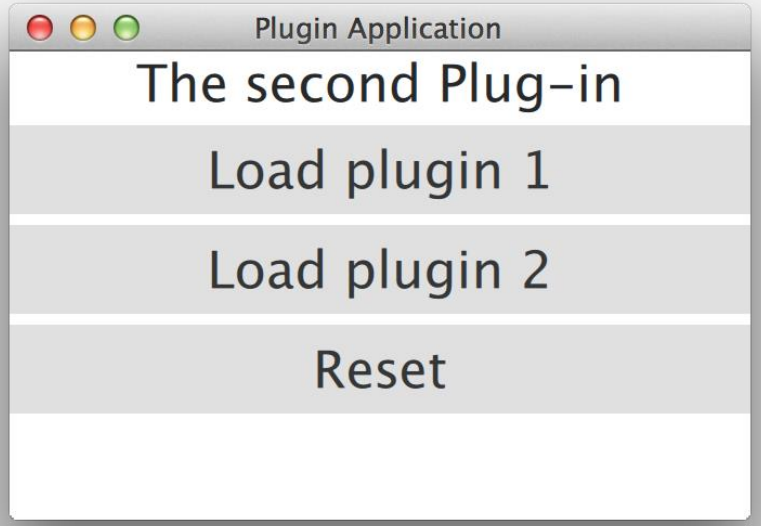
- What are the differences between a shared library and plugin?
- Is it possible to load shared libraries without linking them using `LIBS` variable in the .pro file?
- How plugins can be used in a statically linked program?
- What are plugin low-level and high-level APIs?
- How, when, and from which location does an application load plugins?

Summary

- Application engines can be isolated from the GUI by implementing shared libraries or plugins
  - Shared libraries share data and functionality
  - Plugins provide interface implementations
- Shared libraries are typically loaded, when an application starts
  - Possible to load and unload libraries and resolve symbols dynamically
- Plugins are loaded when requested
  - Exported symbols are defined by the interface
- Programs search for plugins in pre-defined locations
  - If high-level API is used, a Qt class typically takes care of loading plugins
  - If low-level API is used, plugins are loaded by the developer

Lab – Custom Plugin

- Define a custom interface
- Implement one or more plugins, implementing the interface
- Complete the skeleton program, loading the plugin
- Further implementation details in readme.txt



Contents

Objectives

Learn…

› …writing and executing unit tests with Qt Test

› …testing signals and slots

› …benchmarking code blocks

Qt Test Module Features

Lightweight	Consists of about 6000 lines of code and 60 exported symbols
Self-contained	Requires only a few symbols from the Qt Core library for non-GUI testing
Rapid testing	Needs no special test-runners; no special registration for tests
Data-driven testing	A test can be executed multiple times with different test data
Basic GUI testing	Offers functionality for mouse, touch, and keyboard simulation
IDE friendly	Outputs messages that can be interpreted by Visual Studio and KDevelop
Thread-safety	The error reporting is thread safe and atomic
Type-safety	Extensive use of templates prevent errors introduced by implicit type casting
Easily extendable	Custom types can easily be added to the test data and test output

Creating a Unit Test

- A test project wizard provided in Qt Creator
- Create two subprojects using `SUBDIRS`
  - The actual project
  - An adjacent test project
  - Example project .pro file

```
SUBDIRS = \
interestingProject.pro \
interestingProject_tst.pro
```

- Write test cases, while you develop the code
  - Test-driven development
  - Do not mix test code and project code

Test Cases

```
class ExTestTest : public QObject
{
    Q_OBJECT
public:
    ExTestTest();
private:
    // The two optional functions below may be private slots as well
    void initTestCase(); // Called before any test has been executed
    void cleanupTestCase(); // Called after all the test have been executed

    // Test case functions must be private slots
    // They are executed in the declaration order
private Q_SLOTS:
    void testCase1();
    void testCase2();

    // Two optional test case functions executed differently
    init(); // Called before each test case void
    void cleanup(); // Called after each test case
};
```

Test Project

- Macros `QTEST_MAIN` and `QTEST_APPLESS_MAIN` define the `main()` function
- `I`nstantiate the test class and execute all the test cases
- All the test case functions are run on that same instance
- Macros suggest that each test class is compiled and linked to one executable

Running Tests
Command Line Options

- Output format
  - `txt`
  - `csv`
  - `xml`
  - `xunitxml`
- Verbosity
  - `silent` – failure and fatal errors only
  - `v1`- start of each test function
  - `v2` – each `QVERIFY/QCOMPARE/QTEST`
- Testing options
  - `functions` – list test functions
  - `datatags` – list data tags
  - `eventdelay` – default delay in mouse and keyboard simulation in ms
  - `nocrashhandler` – useful for debugging crahses

```
./test
./test testCase1
./test testCase1:testData1
```

Test Results

- Test cases may be
  - Skipped, if a tested feature is not present in the current configuration
    - `QSKIP("This test requires feature X")`
  - or blacklisted, if test cases are skipped in some platform, OS, toolchain, distribution or architecture
    - `android`
    - `ios`
    - `winrt`
    - `[testSomethingNotPresentOnMobilePlatforms]`

```
PASS : MyDataTest::initTestCase()
PASS : MyDataTest::myTestCase
PASS : MyDataTest::cleanupTestCase()
Totals: 4 passed, 0 failed, 0 skipped, 0 blacklisted
********* Finished testing of MyDataTest *********
```

Notes

- If an exception is thrown, rest of the test functions are not executed
  - May produce misleading results
  - There may be more test functions skipped than reported
- You can combine more than just one test class together
  - Do not use `QTEST_MAIN` or `QTEST_APPLESS_MAIN`
- There is no new instance for each invocation of a test function
  - As you may have used to have in other test frameworks
- Executable returns (`QTEST_MAIN/QTEST_APPLESS_MAIN`) a fail count by default, which is useful for scripts

Test Macros

- Several useful macros available to write test cases
  - Defined in `QTest` name space

`VERIFY(condition)`

`QTRY_VERIFY(condition)`

`QTRY_VERIFY_WITH_TIMEOUT(condition, timeout)`

`VERIFY2(condition, message)`

- An additional message is recorded into the test log if the condition is not true

`QTRY_VERIFY2_WITH_TIMEOUT(condition, message, timeout)`

`QCOMPARE(actual, expected)`

- Records the actual and expected values into the test log if they do not match

`QTRY_COMPARE(actual, expected)`

`QFAIL(message)`

- Fails the test case, supposed to be used within the logic of a test function

`QWARN(message)`

- Can be used to record a message to the test log

Enabling Verbose Output

- `QCOMPARE` macro uses `QTest::toString()` functions to output verbose data of different argument types in case the comparison fails
  - Useful to add support for relevant custom types by adding specializations of overloads

```
namespace MyNamespace {
    char *toString(const MyPoint &point)
    {
        // bring QTest::toString overloads into scope:
        using QTest::toString;
        // delegate char* handling to QTest::toString(QByteArray):
        return toString("MyPoint(" +
            QByteArray::number(point.x()) + ", " +
            QByteArray::number(point.y()) + ')');
    }
}
```

Data-Centric Test Case

- Possible to define data sets (tables) for tests
  - Data defined in `testCase_data()` functions

```
#include <QtTest/QTest>

class MyDataTest : public QObject
{
    Q_OBJECT

private slots:
    void myTestCase_data();
    void myTestCase();
}
```

Providing Test Data

- Add columns with arguments and expected results
- Add rows of data
  - The argument of the `newRow()` function defines a tag, which can be used to include a row in the test
  - By default all rows will be included

```
void MyDataTest::myTestCase_data()
{
    // test data table with two columns
    QTest::addColumn<int>("integer_input");
    QTest::addColumn<QString>("result");

    // test data
    QTest::newRow("1st row") << 1 << "yahoo";
    QTest::newRow("2nd row") << 2 << "hello";
    QTest::newRow("3rd") << 3 << "yeah";
}
```

Feeding Test Data for Test Case

- Test function is called multiple times (number of rows)
- Test function produces a single pass/fail result

```
void MyDataTest::myTestCase()
{
    QFETCH(int, integer_input);
    QFETCH(QString, result);
    QCOMPARE(myTestedFunction(integer_input),result);
}
```

QTest GUI Testing Support

- `QTest` class can be used to:

1. Simulate key events
2. Simulate key presses: Up and Down
3. Simulate mouse events: Click and Move
4. Simulate mouse presses: Up and Down
5. Simulate sequences of touch events
6. Check the current test function or data to initialize or cleanup something
7. Convert values of various types into strings

Simulate Key/Mouse/Touch Events

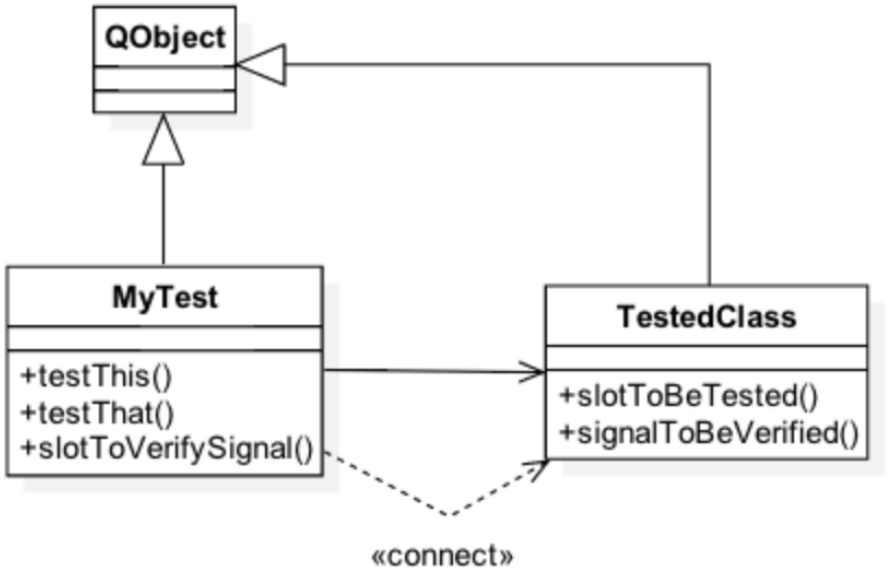
```
void MyTest::init()
{
    _tested = new QLineEdit("default");
}

void MyTest::testcase()
{
    QString defaultValue("default");
    QString input("abc");
    QTest::keyClicks(_tested, input);
    // mouseClick(), mouseDClick(), mouseMove(), touchEvent()
    QString result(_tested->text());
    QString expected(defaultValue + input);
    QCOMPARE(result, expected);
}
```

Testing Asynchronous Functions

- No mock objects exist in QTestLib
  - Order and quantity of calls of slots/functions need to be recorded
  - `QSignalSpy`
- The challenge is not to fall out the test slot function before verifying signals emitted by the tested class
  - `QTest::qWait()`?
  - How long to wait?
- `QSignalSpy::wait()`
  - Starts an event loop
  - Waits until a signal or timeout occurs

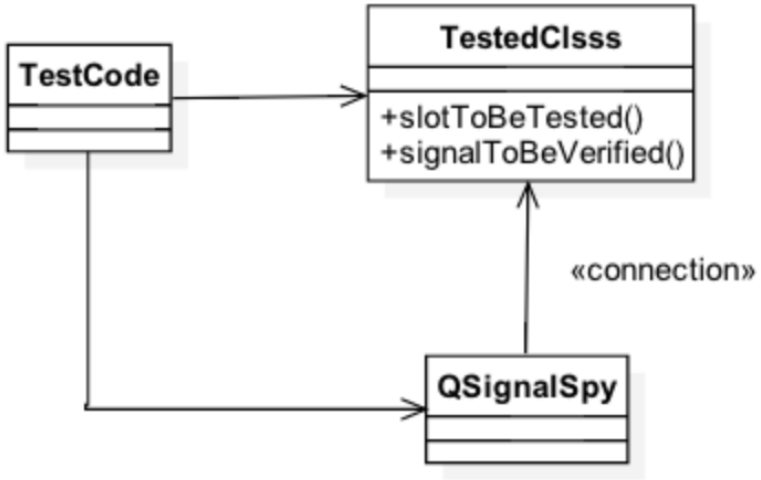
Asynchronous Signals

- A signal can be emitted asynchronously after a tested slot is called
- Test code needs to connect to that signal to receive it
- The connected slot needs to verify the signal



QSignalSpy

- Can be used to record calls of a single signal
- Records the values of the call



Example Test with QSignalSpy

```
void MyTest::testStart()
{
    MyTimer timer;
    QSignalSpy spy(&timer, &MyTimer::ownTimerTimeout);

    QVERIFY(spy.isValid()); // valid signal
    QVERIFY(spy.isEmpty()); // no calls, precondition

    const unsigned long int Period = 2;
    timer.ownStartTimer(Period);
    QVERIFY(spy.wait(Period * 10));

    const int result = spy.count();
    QCOMPARE(result, 1); // one call expected
    QList<QVariant> theCall = spy.takeFirst();
    QVERIFY(theCall.isEmpty()); // no parameters
}
```

Recorded Calls

- `QSignalSpy` is `QList<QList<QVariant > >`
- All parameters of calls can be accessed and verified through `QList`
- Values are stored as `QVariants`
  - `QVariant` provides converter functions
  - For example, `QVariant::toInt()`

QBENCHMARK Macro

- Simple to use
- Code block may be iterated
  - Affects to the test design and implementation
- Several measurement back-ends possible
  - Wall time, CPU tick count, valgrind/callgrind, event count
  - Availability depends on the platform
- Sometimes more straightforward to use `QTime::elapsed()` function

Benchmarking

```
QBENCHMARK { // or QBENCHMARK_ONCE
    … code to be measured …
}
```

```
PASS : Container_perfTest::initTestCase()
PASS : Container_perfTest::testCase1()
RESULT : Container_perfTest::testCase1():
0.50 msecs per iteration (total: 64, iterations: 128)
PASS : Container_perfTest::testCase2()
RESULT : Container_perfTest::testCase2():
0.54 msecs per iteration (total: 70, iterations: 128)
```

Questions and Answers

- What kind of tests can be written with Qt Test?
- How tests are executed?
- Name at least five macros, which can be used in writing tests. How do these macros work?
- Explain differences between `QTest::qWait()` and `QSignalSpy::wait()`. Which one would you prefer in testing signals?
- What should you take into account, when benchmarking code with `QBENCHMARK`?

Summary

- Qt Test allows you to write unit tests for your classes
- Qt Test provides a framework to implement and execute test functions
- Several macros, such as `QVERIFY`, `QCOMPARE`, `QSKIP`, `QFETCH`, `QBENCHMARK`, are available to implement test cases
- Test data can be provided in a table format
  - Data rows can be selected using the command line options
- GUI events can be simulated in tests
- Emitted signals and their parameters can be tested with `QSignalSpy`

Lab – Benchmarking Iterators

- You are provided with a console program, using Java-style and STL-style iterators on an associative container

- Create a test project with test functions, which benchmark the iterators
  - Java-style iterator
  - STL-style const iterator
  - Java-style mutable iterator
  - STLstyle non-const iterator
  - `Q_FOREACH` vs. range-based loop

- You need to reset the iterator, if the benchmarked code is executed more than once

Contents

- Database Connection
- Driver Plugins
- SQL Queries
- Database Item Models
- Transactions

Objectives

Learn…

› …how to manage database

› …how to create and execute SQL queries

› …mapping query results into item models

Qt Database Module

- Qt contains cross-platform and database-independent SQL APIs
- All database-specific code for accessing the database is hidden behind a special driver plug-in
- In order to use the SQL support, add `QT += sql` to your **.pro** file

Database Connection

- Provides an interface to a database using database-specific drivers
  - Any number of connections to one or more databases supported
- `QSqlDatabase::addDatabase(const QString &type, const QString &name)` returns an object for the database connection
  - Type defines the driver to be used
  - Connection is identified by a name – using an existing connection name replaces the old one
  - Connection may only be used in the thread, where it was created
- If no connection name is given, the default connection is used in SQL queries later on
- `QSqlDatabase::connection(const QString &name)` returns a connection object, provided the connection has been previously added

Supported Connection Types

- Due to license incompatibilities with the GPL, not all of the plugins are provided with open source versions of Qt

Driver type	Description
QDB2	IBM DB2, v7.1 and higher
QIBASE	Borland Interbase Driver
QMYSQL	MySQL Driver
QOCI	Oracle Call Interface Driver
QODBC	ODBC Driver
QPSQL	PostgreSQL v7.3 and higher
QSQLITE	SQLite version 3 or above
QSQLITE2	SQLite version 2
QTDS	Obsolete, superseded by ODBC

Connecting to a Database

- Connecting options can be provided with connection member functions
  - `setDatabaseName()` // May be a name, Oracle TNS name, MS Access .mdb file name
  - `setHostName()`
  - `setUserName()`
  - `setConnectOptions()`
  - `setPassword()`

- Before any queries can be done, the database connection is opened using `open()`
  - Returns `true` if a connection could be established or `false`, if something went wrong

Connection Example

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
db.setHostName("bigblue");
db.setDatabaseName("flightdb");
db.setUserName("acarlson");
db.setPassword("1uTbSbAs");

bool ok = db.open();

if (!ok)
    qFatal() << "Error opening database: " << db.lastError();
```

Connection Options

- Database-specific options
  - Read only access
  - SSL connection required
  - Login timeout
- Set with semicolon-separated key=value pairs
- MySQL
  - `db.setConnectOptions("CLIENT_SSL=1;CLIENT_IGNORE_SPACE=1");`
- ODBC
  - `db.setConnectOptions("SQL_ATTR_ACCESS_MODE=SQL_MODE_READ_ONLY)`

Error Handling

- In case `QSqlDatabase::open()` fails, error messages and error codes can be obtained from the object returned by the method `QSqlDatabase::lastError()`

- The error object contains, among others, the following methods:
  - `driverText()`,
  - `databaseText()`,
  - `text()` (a concatenation of the previous two functions),
  - `type()` (driver error number), and
  - `number()` (database error number)

- Note that the text returned from `databaseText()` is most likely not localized

Database Tables, Records, and Features

- Function `tables()` returns the list of tables and views
- Function `primaryIndex()` returns a table's primary index
- To get meta-information about table's fields, call `record(const QString &tableName)`
  - Returns `QSqlRecord`, containing table fields in undefined order
- To check, whether the database driver supports some feature, use `hasFeature()` function
  - `hasFeature(QSqlDriver::QuerySize)`
- Available drivers can be queried using `drivers()`

Driver Plug-ins

- For non-supported database types custom driver plugins can be implemented
  - Not necessary to implement the plugin at all

```
QSqlDatabase::registerSqlDriver("CUSTOMDRIVER",
                                new QSqlDriverCreator<CustomDriver>);
```

1. Derive a class from `QSqlDriver` and implement the pure virtual functions
   - Provides concrete implementation of `QSqlDatabase` functions
2. Derive a class from `QSqlResult` and implement the pure virtual functions
   - Provides concrete implementation of `QSqlQuery` functions

- Notice that the source code for existing drivers is provided in any Qt release – use those as examples!
  - Plenty of more information available in Qt Assistant, as well

Driver Plug-ins

```
class QSQLiteDriverPlugin : public QSqlDriverPlugin
{
    Q_OBJECT Q_PLUGIN_METADATA(IID "org.qt-project.Qt.QSqlDriverFactoryInterface"
                               FILE "sqlite.json")
public:
    QSQLiteDriverPlugin();
    QSqlDriver* create(const QString &) Q_DECL_OVERRIDE;
};
class QSQLiteDriver : public QSqlDriver
{
    Q_OBJECT
public:
    explicit QSQLiteDriver(QObject *parent = 0);
    bool hasFeature(DriverFeature f) const Q_DECL_OVERRIDE;
    bool open(const QString &db, const QString &user, const QString &password,
              const QString &host, int port, const QString &connOpts);
         // sqlite3_open_v2(file, sqlite3Struct, openMode)
    QSqlResult *createResult()
    QStringList tables(QSql::TableType) const Q_DECL_OVERRIDE;
    QSqlIndex primaryIndex(const QString &table) const Q_DECL_OVERRIDE;
```


qtbase/src/plugins/sqldrivers/sqlite

QSqlQuery

- Wrapper to `QSqlResult` in the driver
  - Supports any query, supported by the driver
  - An optional argument can be given to the constructor, specifying which database to use
- Run a query by calling `exec()`
  - `size()` reports how many rows were matched by a select query
  - I -1 if the number of rows can not be determined
  - `numRowsAffected()` tells how many rows were affected by a non-select query, say, an update query

```
QSqlQuery query;
if (!query.exec("SELECT name FROM author"))
    // ...
```

SQL Queries

- In case of a select statement, the result can be iterated over using `QSqlQuery::next()`
  - This method returns `true` as long as there are more records available
- The value of the records is fetched using `QSqlQuery::value(int)`, which returns a `QVariant`
- For navigation you can use:
  - `QSqlQuery::first()`,
  - `QSqlQuery::last()`,
  - `QSqlQuery::prev()`
  - `QSqlQuery::seek(int)`

```
QSqlQuery query("SELECT country FROM artist");
while (query.next()) {
    QString country = query.value(0).toString();
    doSomething(country);
}
```

- `QSqlQuery::lastError()` can be used to query for error messages

Prepared Queries

- May speed up inserting a large number of records
- If the database does not support prepared queries Qt will translate the query into an ordinary query
- Two kinds of prepared queries:
  - named bindings
  - positional bindings

Bindings

```
// Named bindings
QSqlQuery query;
query.prepare("INSERT INTO employee (id, name, salary) VALUES (:id, :name, :salary)");
query.bindValue(":id", 1001);
query.bindValue(":name", ”Employee 1");
query.bindValue(":salary", 10000000);
query.exec();
```

```
// Positional bindings
QSqlQuery query;
query.prepare("INSERT INTO employee (id, name, salary) VALUES (?, ?, ?)");
query.addBindValue(1002);
query.addBindValue(”Employee 2");
query.addBindValue(10000001);
query.exec();
```

Database Item Models

- `QSqlQueryModel` wraps a `QSqlQuery` in a `QAbstractItemModel`
  - The result set of the query can be used with the model/view framework
- The titles displayed in views are the column names from the database
  - Can be changed using `QSqlQueryModel::setHeaderData()`
- `QSqlTableModel` wraps *a single table* in a model, and does therefore allow editing the items
  - Create an instance of `QSqlTableModel`, and call `setTable()` specifying the table to use
  - Optionally call `setFilter()` specifying a WHERE part of a SQL query
  - Optionally call `setSort()` specifying column number and sort direction
  - Call `select()` to execute the query
- `QSqlRelationalTableModel` is `QSqlTableModel` subclass with a foreign key support

QSqlTableModel vs. QAbstractItemModel

- Possible to access the table programmatically using the methods of `QAbstractItemModel`
- `QSqlTableModel` adds a few methods for convenience
  - `record()`, `setRecord()` and `insertRecord()` all work with instances of `QSqlRecord`
  - All refer to rows in the table rather than `QModelIndexes`
- `QSqlRecord` is a simple container for records containing methods like
  - `setValue(int index, QVariant value)`,
  - `setValue(QString name, QVariant value)`, and similar
  - `QVariant value(...)` methods

QSqlTableModel

```
for (int row = 0; row < model->rowCount(); ++row) {
    QSqlRecord record = model->record(row);
    double price = record.value("price").toDouble();
    price *= 1.1;
    record.setValue("price", price);
    model->setRecord(row, record);
}

model->submitAll();
```

Commit

- Using `setEditStrategy()` it is possible to specify when changes made in the GUI should be committed to the database

- Edit strategies
  - `OnFieldChange` – Data will be saved as soon as you start editing a new cell
  - `OnRowChange` – Data will be saved when you start editing a new record (changes can be discarded by calling `revert()`)
  - `OnManualSubmit` – Data will only be saved when you call `submitAll()` (changes can be discarded with `revertAll()`)

- Be careful with `OnFieldChange`:
  - Performance can drop significantly compared to using the other editing strategies
  - If you modify a primary key, the record might slip through your fingers while you are trying to fill it

Editable Queries

- Without modifications `QSqlQueryModel` is read only, while `QSqlTableModel` only works on a single table

- To be able to edit the result of an arbitrary query, override
  - `QAbstractItemModel::setData()` to update the data yourself, and
  - `QAbstractItemModel::flags()` to specify that the table is editable

ex-editable-query

Transactions

- You start a transaction using
  - `QSqlDatabase::transaction()`, and end it using
  - `QSqlDatabase::commit()` or
  - `QSqlDatabase::rollback()`
- The above methods return `true` if the action succeeded
- Transaction requires support from the database – check for this using
  - `QSqlDriver::hasFeature(QSqlDriver::Transactions)`

Questions and Answers

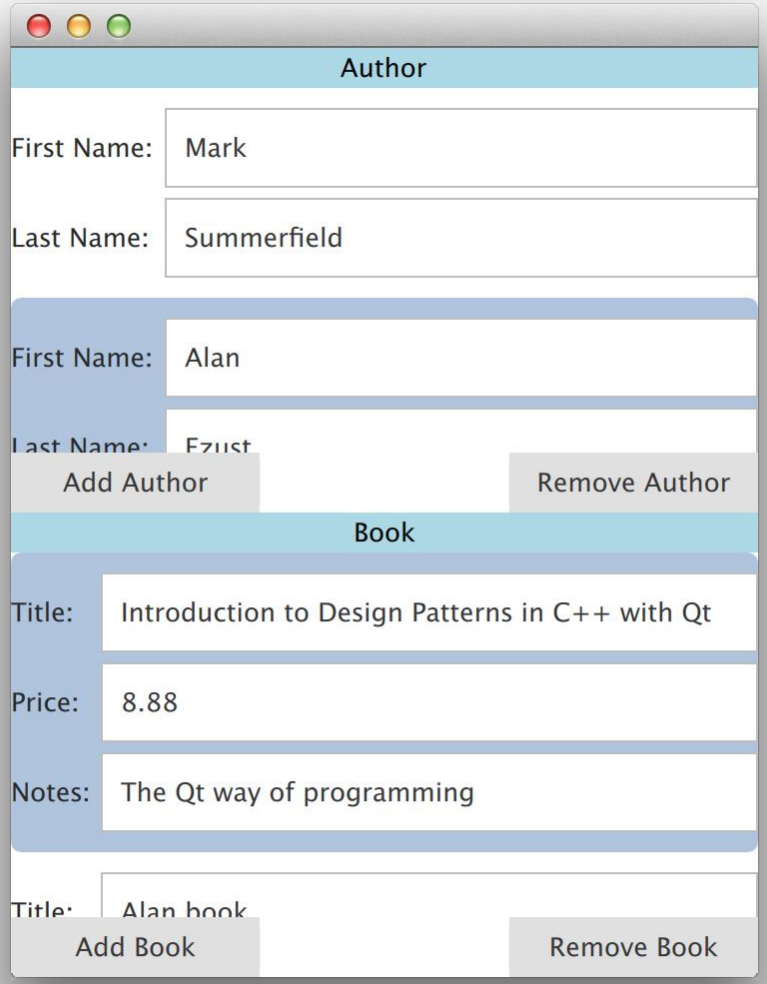
- What is the role of `QSqlDatabase`?
- Can you share database connection between threads? Justify.
- How can you check, whether a required feature can be used with an existing database driver?
- What kind of SQL queries are supported by `QSqlQuery`?
- What and named and positional value bindings? Is it beneficial to use them?
- What kind of item models can be used with databases?
- What should be taken into account performance wise when using `QSqlTableModel`?
- Does Qt support transactions?

Summary

- Qt has support to make SQL queries to the database
  - Intuitive Qt-style APIs provided for composing the queries
- Queries are made using an open database connection
  - Several connections may be opened to the same database
  - Connections are value types, which cannot be shared between threads
- Qt database system is based on the model/view framework with three layers
  - Database technology -based drivers – connection and query objects are wrappers to the driver object
  - Model classes – mapping query result or a single table
  - View classes – Widgets or QML types

Lab – Bookstore

- Author table in upper view
- Book table in lower view
  - Only books from current author shown
- Follow these steps (more details in readme.txt)
  - Setup the author table (`QSqlTableModel`)
  - Setup the proxy tables to map columns
  - Setup book table with `QSqlQueryModel`
  - Provide edit support for both tables
- Optional
  - Support add/delete rows



Contents

Objectives

Learn…

› …what services Qt Multimedia module provides

› …audio and video playback and recording

› …accessing video pixel data

Qt Multimedia Features

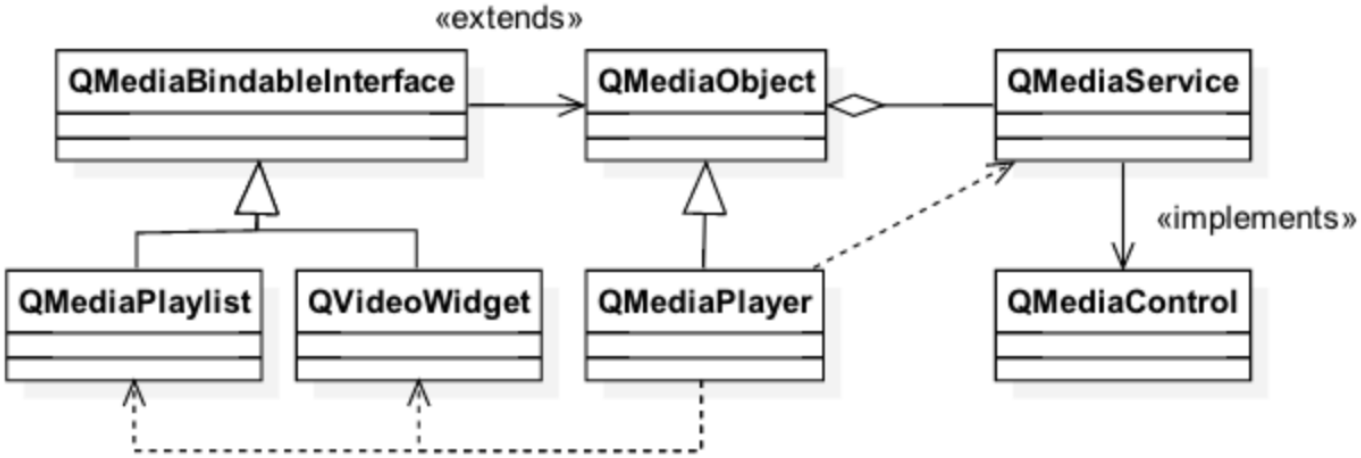
- Media file playback
  - `QMediaPlayer`
  - `QMediaPlaylist`, `QMediaContent`
  - `QVideoWidet`
- Audio device access
  - `QAudioDeviceInfo`
  - `QAudioInput`, `QAudioOutput`
  - `QAudioFormat`
  - `QAudioBuffer` – audio media stored in memory for processing
- Low latency sound effects
  - `QSound` – plays .wav files
  - `QSoundEffect`
- Camera and view finder
  - `QCamera`
  - `QAbstractVideoSurface`
  - `QAbstractVideoFilter` – for QML
  - `QVideoFrame`
- Audio buffer and video frame monitoring
  - `QAudioProbe`
  - `QVideoProbe`
- FM radio
  - `QRadioTuner`
  - `QRadioData`
- 3D positional audio – Qt Audio Engine

Multimedia Architecture

- High-level multimedia classes derive from `QMediaObject`
  - `QMediaPlayer`, `QAudioRecorder`, `QCamera`, `QRadioTuner`
- Media object
  - Provides access to meta-data: title, language, copyright, publisher
  - Provides internally a media service object, which actually implements the multimedia service
  - Allows binding helper objects, implementing `QMediaBindableInterface`
- `QMediaService` implements one or more media control interfaces
  - E.g., `QAudioRecorderComtrol`, `QCameraZoomControl`, `QRadioDataControl`
- Helper objects extend media object functionality
  - `QMediaPlaylist`, `QRadioData`, `QVideoWidget`

Audio and Video Playback

- `QMediaPlayer` is a media object using internally a certain media service
  - Extended with `QMediaPlaylist` and `QVideoWidget` helper classes



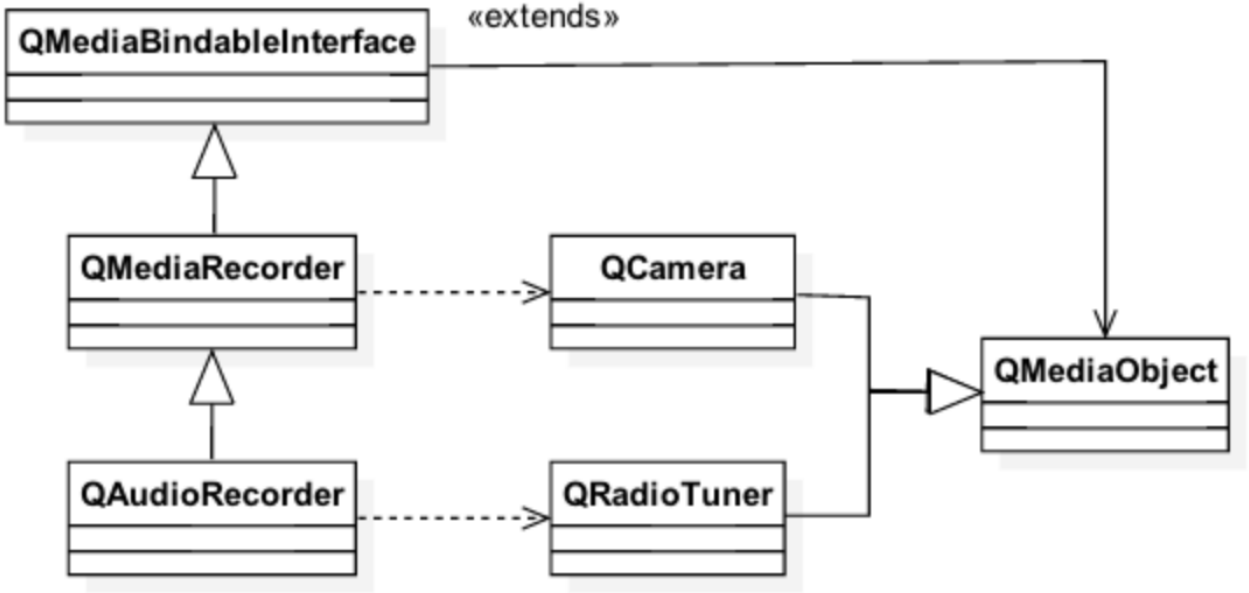
Audio and Video Playback

```
m_player = new QMediaPlayer;
m_playlist = new QMediaPlaylist(m_player);
m_playlist->addMedia(QUrl("video.mp4"));
m_widget = new QVideoWidget();
m_player->setVideoOutput(m_widget);
m_playlist->setCurrentIndex(1);
m_player->play();
```

```
FileDialog { id: filedialog
    title: qsTr("Please choose a media file")
    folder: shortcuts.home
    onAccepted: mediaplayer.source = filedialog.fileUrls[0];
}
VideoOutput { id: videooutputvideo
    anchors { top: parent.top; bottom: toolbar.top }
    width: parent.width
    source: mediaplayer
}
MediaPlayer { id: mediaplayer }
```


ex-avplayer-widgets and ex-avplayer-qml

Audio and Video Recording



Audio and Video Recording

- `QAudioRecorder` allows recording and compressing audio data

```
audioRecorder = new QAudioRecorder;
QAudioEncoderSettings audioSettings;
audioSettings.setCodec("audio/amr"); audioSettings.setQuality(QMultimedia::HighQuality);
audioRecorder->setEncodingSettings(audioSettings);
audioRecorder->setOutputLocation(QUrl::fromLocalFile("test.amr"));
audioRecorder->record();
```

- `QMediaRecorder` allows recording video
  - Set the source in the constructor (a camera or a radio tuner)
  - Set audio settings as above
  - Start recording

Other Audio Classes

- `QSoundEffect`
  - Low latency WAV format sound effects
  - Volume, mute, and number of loops may be controlled
- `QAudioProbe`
  - Monitor played or recorded audio data
  - Any media object may be used as a source
- `QAudioOutput` and `QAudioInput`
  - Raw audio data output and input
  - Available HW determines what audio input and outputs are available
- `Qt Audio Engine`
  - QML module for providing 3D positional audio playback and content management
  - Wave files are organized into discrete Sound instances, which are grouped and controlled using categories

Accessing Low Level Video Frames

- Useful when accessing barcodes or applying fancy effects to the frames
- Set the video output of the media player to your custom surface

```
class MyVideoSurface : public QAbstractVideoSurface
{
    QList<QVideoFrame::PixelFormat> supportedPixelFormats(
         QAbstractVideoBuffer::HandleType handleType =
         QAbstractVideoBuffer::NoHandle) const {
            Q_UNUSED(handleType);
            // Return the formats you will support
            return QList<QVideoFrame::PixelFormat>() << QVideoFrame::Format_RGB565;
    }
    bool present(const QVideoFrame &frame)
    {
        Q_UNUSED(frame);
        // Handle the frame and do your processing return true;
    }
```


ex-custom-video-widget

FM Radio

- Radio tuner + access to RDS
- `QRadioTuner`
  - Media object
  - Frequency control
  - Stereo mode control
  - Provides access to `QRadioData`
- `QRadioData`
  - Station name
  - Station id
  - Radio text

```
radio = new QRadioTuner;
connect(radio, SIGNAL(frequencyChanged(int)), this,
                SLOT(freqChanged(int)));
if (radio->isBandSupported(QRadioTuner::FM)) {
    radio->setBand(QRadioTuner::FM);
    radio->setFrequency(yourRadioStationFrequency);
    radio->setVolume(100);
    radio->start();
}
```

Questions and Answers

- What are media objects, media services, and media controls and how are they related to each other?
- In which ways is it possible to play back audio using Qt multimedia?
- Which media codecs are supported by Qt?
- How video frames can be manipulated?
- How would you provide data to computer vision libraries, such as OpenCV?

Summary

- Qt Multimedia provides a rich set of multimedia features
  - Audio and video playback and recording
  - Low-latency sound effects
  - Manipulation of raw audio data and video frames
  - FM radio
  - 3D audio
- Features are used with media objects
  - Media objects use media services, which actually implement the requested services – possibly using the underlying platform libraries
  - Media objects may be extended with helper objects, like media player can be extended with a play list and video widget
- Qt allows monitoring and changes audio buffers and video frames

Contents

- Qt Speech
- Text to Speech

Objectives

Learn…

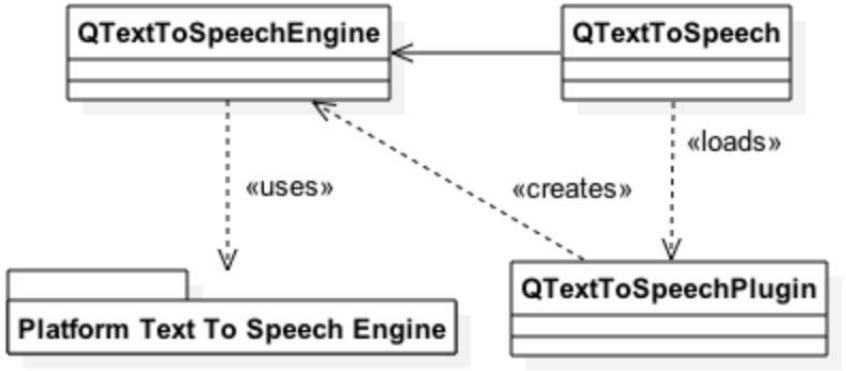
› …Qt APIs to text to speech engines

Qt Speech

- Currently supports Text to Speech (TTS) only
  - `QT += texttospeech`
- Uses platform APIs to access text to speech engines
  - In Linux, Speech Dispatcher library needed
- Speech recognition module under development

Text to Speech

- All functionality implemented in the backend engine
- QTextToSpeech is a wrapper to the engine
- Query the engines with `availableEngines()`
- Select the engine in the constructor
- Query and set the
  - language: `QVector<Qlocale> availableLocales()`
  - voice: `QVector<QVoice> availableVoices()`



Voice Control

- `QVoice` controls the voice
  - Age (Child, Teenager, Adult, Senior, Other)
  - Gender (Male, Female, Unknown)
  - Name
- QTextToSpeech allows controlling
  - the rate and pitch in the range [-1.0, 1.0]
  - the volume in the range range [0, 100]
- Text is synthesized asynchronously using function `say(const QString &)`

Text To Speech

```
QStringList engines = QTextToSpeech::availableLocales();
// Let the user to select the engine

m_speech = new QTextToSpeech(engine, this);
QVector<QLocale> locales = m_speech.availableLocales();
// Let the user select the language

m_speech.setLocale(locale);
m_speech.setRate(rate);
m_speech.setPitch(pitch);

QVector<QVoice> voices = m_speech.availableVoices();
// Let the user select the voice
m_speech.setVoice(voice);

m_speech.say("Hello World");
```

Summary

- Qt Speech module supports access to speech synthesizer engines
- `QTextToSpeech` is a simple wrapper, which loads the engine and calls its members to synthetize speech
- `QTextToSpeech` allows a user to choose a language and voice
  - Volume, rate, and pitch can be controlled as well
- Voices are provided by the platform
  - Voice parameters include gender and age
  - Voices are identified by a string name

Contents

Objectives

Learn…

- …XML parsing options
- …XML parsing with XML stream reader
- …XQuery in Qt
- …JSON parsing

XML APIs

Qt provides three different means of accessing XML data:

- SAX (simple API for XML, version 2)
  - Provides a sequential view on the data using call backs
- Stream Reader/Writer
  - Also a sequential view, but control is in the application
  - Makes it easier to write recursive descent parsers
- DOM (document object model, level 1 and 2), which provides a tree view on the data
- SAX and DOM APIs are deprecated and not covered here

XML Parsing with Stream Reader

- `QXmlStreamReader` provides fast and efficient way to parse XML
  - Well-formed XML 1.0 parser
- Small memory usage
  - XML data is parsed by pulling tokens using `TokenType readNexr()`
  - Only current token kept in memory
  - String data reported with `QStringRef`
- Incremental parsing
  - Data read in chunks
  - `PrematureEndOdDocumentError` reports the document was not fully parsed
  - Possible to resume once data is available

QXmlStreamReader

- Data may be read from any `QIODevice`, string, byte array or char pointer
  - `QXmlReader reader(tcpSocket);`
  - Function `addData()` adds more data for the reader
- Data is read in the loop using `readNext()` or `readNextStartElement()`

```
while (!reader.atEnd()) {
            while (reader.readNextStartElement())
```

  - Irrelevant elements may be skipped with `skipCurrrentElement()`
- Fetch element data
  - `name()` – element name
  - `text()`, or `readElementText()` – returns everything till the matching end token as text
  - `attributes.value("attributeName")`

QXmlStreamReader

- Error handling
  - If `readNext()` reports an error, it returns `EndDocument`, which means `atEnd()` returns true

```
if (reader.hasError)
      Error error = reader.error();
```

- You can also signal an error yourself using `raiseError(QString msg)`

```
if (reader.readNextStartElement()) {
    if (reader.name() != ”bookmarks” || xml.attributes().value("version") != "4.2”)
        xml.raiseError(QObject::tr("The file is not a bookmarks version 4.2 file."));
    else
        // Continue handling
}
```

QXmlStreamWriter

- Allows you to write XML in a streaming fashion, using high level functions
- Data are written using methods like `writeStartDocument()`, `writeStartElement()`, `writeEndElement()`, `writeAttribute()`, `writeCharacters()`
- Specify the device to write to using `setDevice(QIODevice*)`
- To get human readable XML generated (e.g. new lines in place), call `setAutoFormatting(true)`

XQuery and XPath

- An improved way of working with XML where your level of abstraction is higher than regular XML
- Allows you both to parse complex XML, and to generate XML based on another XML file
- You can:
  - read data from XML
  - filter and sort the data, make search and select tasks
  - write the result to a new XML document
  - create a completely new XML document
- The XQuery-related classes can be found in the Qt XmlPatterns module
  - Also used in `XMLListModel` in QML
- Queries can be run from within your C++ code or using a separate command line utility application (called xmlpatterns)

XQuery and XPath

- Write XQuery statements in your code or in a separate text file (e.g. `myqueries.xq`)

```
<selectedcars>
   doc("cars.xml")/car[engine = "V8"]
</selectedcars>
```

- Starting from the document (root) node of cars.xml
- Pick each `<car>` element anywhere in the document where the `<engine>` child element's value is "V8"

```
<?xml version="1.0"?>
<cars>
   <car>
      <make>Trabant</make>
      <model>Convertible</model>
      <engine>V8</engine>
   </car>
```

XQuery in Qt

- `QXmlQuery` executes queries in the XQuery language
- A query is added with `QXmlQuery::setQuery()` and evaluated with the `QXmlQuery::evaluateTo()` methods
- Queries can be evaluated to `QStringList`, `QXmlResultItems` or `QAbstractXmlReceiver`

Evaluating to QStringList

- Evaluating to QStringList is possible only if the query evaluates to a sequence of string values
- Example - read the text element from the paragraphs of index.html and puts the result into a `QStringList`

```
QXmlQuery query;
query.setQuery("doc('index.html')/html/body/p/string()");
QStringList result;
query.evaluateTo(&result);
```

Evaluating to QXmlResultItems

- `QXmlResultItems` is a sequence of `QXmlItems`
  - A `QXmlItem` represents either a node or an atomic value
- A null item means it is invalid
- The query below evaluates to a node, an integer and a string

```
QXmlQuery query;
query.setQuery("<myNode />, 1, 'a string'");
QXmlResultItems result;
query.evaluateTo(&result);
QXmlItem item(result.next());
while (!item.isNull()) {
    // use item
    item = result.next();
}
```

Evaluating to QAbstractXmlReceiver

- `QAbstractXmlReceiver` is an abstract class, acting as a callback interface for query evaluation
  - Can be used to transform the output of a `QXmlQuery`
- Its methods are called when an attribute, start/end element, comment, atomic value is found
- `QXmlSerializer` and `QXmlFormatter` are implementations of this interfaces and can be used to save the query result into an XML file

Evaluating to QXmlSerializer

- `QXmlSerializer`: translates an XQuery sequence to XML and writes the result into a `QIODevice`
- Example, selecting the first paragraph from the html body:

```
QXmlQuery query;
query.setQuery("doc('index.html')/html/body/p[1]");
QXmlSerializer serializer(query, myOutputDevice);
query.evaluateTo(&serializer);
```

- The output is not formatted, for example:
  - `<p><b>First</b> paragraph</p>`

Evaluating to QXmlFormatter

- `QXmlFormatter` can be used to format the result of a query

```
QXmlQuery query;
query.setQuery("doc('index.html')/html/body/p[1]");
QXmlFormatter formatter(query, myOutputDevice);
formatter.setIndentationDepth(4);
query.evaluateTo(&formatter);
```

- Example output with `QXmlFormatter`:

```
<p>
    <b>First</b> paragraph
</p>
```

QAbstractXmlNodeModel

- Modeling non-XML data to look like XML for `QXmlQuery`
- Rather complex to sub-class, `QSimpleXmlNodeModel` often used
- Read data from the file using a node model
  - The model will create the XML nodes
- Create an XML query to read queries from the nodes
- Bind the root query variable and the root node
- Evaluate the query

QAbstractXmlNodeModel

```
QFile queryFile(argv[1]);
QFile chemistryData(argv[2]);
QString moleculeName = argv[3];
QXmlQuery query;
query.setQuery(&queryFile, QUrl::fromLocalFile(queryFile.fileName()));
ChemistryNodeModel myNodeModel(query.namePool(), chemistryData);
QXmlNodeModelIndex startNode = myNodeModel.nodeFor(moleculeName);
query.bindVariable("queryRoot", startNode);
QFile out;
out.open(stdout, QIODevice::WriteOnly);
QXmlSerializer serializer(query, &out);
query.evaluateTo(&serializer);
```

XML Schema

- XML Schema is a W3C standard
  - http://www.w3.org/XML/Schema
  - Qt supports XML Schema 1.0
- Schemas specify the structure and contents of XML documents
  - `QXmlSchema` represents a schema
- Documents are validated against schemas
  - `QXmlSchemaValidator` is used to validate documents

Loading a Schema

- Schemas are represented by Uniform Resource Identifiers (URIs)
- Can use the URI to locate the schema
  - `QXmlSchema::load(QUrl(...))`
  - Uses the URI as a URL
  - The schema will be fetched over the network
- Can be loaded from a `QIODevice` or `QByteArray`:
  - `QXmlSchema::load(device, QUrl(...))`
  - `QXmlSchema::load(bytes, QUrl(...))`
  - The URI passed as a QUrl is optional
- Optional URIs are used to resolve relative URIs in the schema

Loading a Schema from a URL

- Loading a schema from a remote location:

```
QUrl url("http://www.schema-example.org/myschema.xsd");
QXmlSchema schema;
if (schema.load(url))
    qDebug() << "schema is valid";
else
    qDebug() << "schema is invalid";
```

- You must verify that the schema is valid

Loading a Schema from a File

- Sometimes better to cache schemas locally

```
QFile file("test.xml");
file.open(QIODevice::ReadOnly);
QXmlSchemaValidator validator(schema);
if (validator.validate(&file, QUrl::fromLocalFile(file.fileName())))
    qDebug() << "instance document is valid";
else
    qDebug() << "instance document is invalid";
```

- Passing a valid URI helps to resolve references in the schema

Validating a Document

- Documents are also represented using URIs:
- Can use the URI to locate the document
  - `QXmlSchemaValidator::validate(QUrl(...))`
  - Uses the URI as a URL.
  - The document will be fetched over the network.
- Can be read and validated from a `QIODevice` or `QByteArray`:
  - `QXmlSchemaValidator::validate(device, QUrl(...))`
  - `QXmlSchemaValidator::validate(bytes, QUrl(...))`
  - The URI passed as a `QUrl` is optional

JSON

- Format to encode object data in JS
- Six basic types
  - `Bool, double, string, array [], object {}, null`

```
{
    “key1”: “value1”,
    “key2”: “value2”,
    “objectKey”: {
        “key4”: “value4”,
        “key5”: “value5”
    },
    etc.
```

JSON Parsing with QJsonDocument

- Provides APIs to parse, modify, and save JSON data
- Speed optimized binary format that is directly memory map-able and very fast to access
  - `QJsonDocument::fromJson() / toJson()`
  - Parses UTF-8 encoded JSON document to the binary format and back
- The document contains an array or an object
  - `QJsonArray/QJsonObject` classes provide API to parse and modify the content
  - An object contains key-value pairs, where a value can be an array, object or any of the basic types
  - The easiest way to parse arrays or objects is to use iterators

```
QJsonObject jsonObject(document.object());
if (jsonObject.contains("key1")) {
    QJsonValue value(jsonObject.take("key1"));
```

```
void parseObject(const QJsonObject &object, QXmlStreamWriter &writer)
{
    QStringList keys = object.keys();
    for (const QString &key : keys) {
        writer.writeStartElement(key);
        parseValue(object.value(key), writer);
    writer.writeEndElement();
    }
}
void parseValue(const QJsonValue &value, QXmlStreamWriter &writer)
{
    if (value.isArray())
        parseArray(value.toArray(), writer);
    else if (value.isObject())
        parseObject(value.toObject(), writer);
    else if (value.isBool()) {
        if (value.toBool())
            writer.writeCharacters("true");
        else
            writer.writeCharacters("false");
    }
    // and so on for double and undefined types
}
```

Performance

- XML stream reader and writer provide typically always the best performance
  - However, the performance is benchmarked to be slower compared to xmllib, for example (10-50%)
- DOM tree provides rather good performance, because all data access may be done in memory
  - However, slows done with large (>10 MB) XML documents
- SAX performance is the worst and the idea is to make porting easier
  - Now you should port directly using stream reader and writer
- JSON handling is better optimized than DOM tree as an internal binary data format is used

Questions and Answers

- What alternatives are there to parse XML in Qt?
- When would you benefit using XQuery and XPath compared to XML stream reader and writer?
- What makes XML stream reader memory efficient?
- How JSON processing is optimized in Qt?

Summary

- Qt provides four ways for parsing XML
  - XML stream reader and writer
  - SAX parser
  - XML parsing using the DOM tree
  - Parsing with XQuery and QPath
- SAX parser and DOM tree are deprecated and they should be used in legacy code only
- Performance wise XML stream reader / writer provides typically the best performance
  - Typical use case is recursive XML parsing
- XQuery provides more convenient way for parsing than XML stream reader, if only certain data is relevant from the XML document
  - Fetch all elements, where data values satisfy a required condition

Lab – Reading and Writing Xml Keys

- `KeyEngine` class allows storing key-value pairs
- Your task to write XML read/write backends

- XML Format shall be:

```
<?xml version="1.0" encoding="UTF-8"?>
<keys version="1.0”>
   <item key="Key-0">Value-0</item>
   <item key="Key-1">Value-1</item>
   …
   <item key="Key-9">Value-9</item>
</keys>
```

Contents

- SCXML
- QScxmlStateMachine
- Data Models
- Invoking Services

Objectives

Learn…

› …Qt SCXML support

› …essential classes and QML types to access state machines

SCXML

- Allows creating state machines statically (during build time) or dynamically (run-time) from SCXML files
- Both C++ and QML types provided
- Allows clear separation of an application UI and application logic
- Based on the meta-object system
  - State transition can be triggered by a signal
  - Property values may be set and methods may be invoked in states

SCXML Specification Briefly

- http://www.w3.org/TR/scxml/

- States
  - Can be nested, can be parallel
  - May be initial, final or history state
  - May invoke external services, Qt supports only other another SCXML state machine
  - May contain transitions

```
<state id=”state1" initial=”state11">
    <state id=”state11”>
```

- Transitions
  - Triggered by an event
  - May have a condition

```
<transition event=”someEvent" cond="In(’someParallelState')" target=”stateX"/>
```

SCXML Specification Briefly

- States may have executable content inside `<onentry>` and `<onexit>` elements
  - Raise events - `<raise event=”anEvent"/>`
  - Send events to external systems - `<send event=”anEvent" id=”evId" delay=“3s"/>`
    - Event parameter(s) are `QVariant(Map)`
    - In case of error events, `_event.errorMessage` contains a more detailed description of an error
  - Log messages - `<log label="'result'" expr=”1 + 3" />`
  - Execute scripts- `<script src=“scripts.js”> or <script>someVar = cppModelFunction()…</script>`
  - Assign values to a data model - `<assign location=”dataModel_var" expr=“1 + 3” />`

- States and state machines may have 0 or more data models
  - ECMAScript model - `<datamodel><data id=”score" expr="0"/></datamodel>`
  - C++ data model - `datamodel="cplusplus:DataModel:datamodel.h”`
  - Null data model

SCXML Example

```
<?xml version="1.0" ?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
        initial="wrapper" datamodel="ecmascript" name="CalculatorStateMachine">
    <datamodel>
        <data id="long_expr" />
        <data id="short_expr" />
        <data id="res" />
    </datamodel>
    <state id="wrapper" initial="on">
        <state id="on" initial="ready">
            <onentry>
                <send event="DISPLAY.UPDATE" />
            </onentry>
        <state id="ready" initial="begin">
            <state id="begin">
                <transition event="OPER.MINUS" target="negated1" />
                <onentry>
                    <assign location="long_expr" expr="''" />
                    <assign location="short_expr" expr="0" />
                    <send event="DISPLAY.UPDATE" />
```

Creating a State Machine

- Dynamic creation
  - `auto *stateMachine = QScxmlStateMachine::fromFile(”voiceController.scxml");`
  - Alternatively, `QScxmlCompiler` can be used
    - `QScxmlCompiler compiler(xmlStreamReader); // Use stream reader to read the file`
    - `QScxmlStateMachine *stateMac = compiler.compile();`
  - In QML use `StateMachineLoader`

```
StateMachineLoader { // State machine available as property stateMachine
    source: “voiceController.scxml”
}
```

- Static creation
  - Use `qscxmlc` tool to compile the SCXML file
    - `STATECHARTS = voiceController.scxm`
    - `VoiceController voiceController; // Type may be registered as a QML type`

QScxmlStateMachine

- Start and stop the state machine
- Exposes all state machine states as Boolean properties
- Access states
  - `QStringList stateNames(bool compress = true) const`
  - `QStringList activeStateNames(bool compress = true) const`
- Observer state changes and events
  - `QMetaObject::Connection connectToState(const QString &stateName, const QObject *receiver, PointerToMemberFunctio method)`
  - `QMetaObject::Connection connectToEvent(const QString &eventSpec, const QObject *receiver, PointerToMemberFunctio method)`

QScxmlStateMachine

- Submit events
  - `submitEvent(const QScxmlEvent *event)`
  - `submitEvent(const QString &eventName, const QVariant &data)`
  - `cancelDelayedEvent(const QString &sendId)`
- Set the data model and initial values
  - Can be set only once – `setDataModel(QScxmlDataModel *model)`
  - `void setInitialValues(const QVariantMap &initialValues)`

QScxmlStateMachine Example

```
m_machine->start();

// Observer slot may have a Boolean arg to see, if the state is entered or exited
m_machine->connectToState(state, observer, &Observer::notify);

// updateScore has two parameters
// <send event="updateScore">
// <param name="highScore" expr="highScore"/>
// <param name="score" expr="score"/>
m_machine->connectToEvent("updateScore", [this] (const QScxmlEvent &event) {
    const QVariant data = event.data();
    const QString highScore = data.toMap().value("highScore").toString();
}

m_machine->submitEvent("randomEvent");
```

State Machine in QML

```
property StateMachine stateMachine: scxmlLoader.stateMachine

StateMachineLoader { id: scxmlLoader }

EventConnection {
    stateMachine: root.stateMachine
    events: ["playbackStarted", "playbackStopped"]
    onOccurred: {
        var media = event.data.media
        theLog.text = "\nplaybackStarted with data: " + JSON.stringify(event.data)
    }
}
function tap(idx) {
    var media = theModel.get(idx).media
    var data = { "media": media }
    stateMachine.submitEvent("tap", data)
}
```

Data Models

- Base class QScxmlDataModel
  - Model property access methods
    - `bool hasScxmlProperty(const QString &name) const`
    - `bool setScxmlProperty(const QString &name, const QVariant &value, const QString &context)`
    - `QVariant scxmlProperty(const QString &name)`
  - Pure virtual functions for evaluating executable content
    - `virtual QVariant evaluateToVariant(QScxmlExecutableContent::EvaluatorId id, bool *ok) = 0`
- Three subclasses
  - `QScxmlCppDataModel`
  - `QScxmlNullDataModel`
  - `QScxmlEcmaScriptDataModel`

QScxmlCppDataModel

- Macro `Q_SCXML_DATAMODEL` results `qscxmlc` to generate required evaluation functions
  - Makes members accessible in SCXML

```
#include "qscxmlcppdatamodel.h"

class TheDataModel: public QScxmlCppDataModel
{
    Q_OBJECT
    Q_SCXML_DATAMODEL // Results qscxmlc to generate evaluate functions
private: // Note private members
    bool isValidMedia() const; // evaluateToBool
    QVariantMap eventData() const; // evaluateToVariant
    QString media; // evaluateToVariant
};
```

QScxmlCppDataModel

- Allows adding C++ statements in `<script>` elements
- Allows using C++ expressions in `cond` and `expr` attributes

```
<scxml
    xmlns=http://www.w3.org/2005/07/scxml
    datamodel="cplusplus:TheDataModel:thedatamodel.h"
>
    <state id="stopped">
        <transition event="tap" cond="isValidMedia()" target="playing"/>
    </state>
    <state id="playing">
        <onentry>
            <script> media = eventData().value(
                QStringLiteral(&quot;media&quot;)).toString();
            </script>
```

Invoking Services

- Only other SCXML state machines can be invoked as services
  - Allows having a state machine inside a state machine
- Accessible with
  - `QVector<QScxmlInvokableService *> QScxmlStateMachine::invokedServices() const` or
  - `InvokedServices` in QML
- Same features as the outer state machine

Summary

- Qt SCXML allows creating state machines from XCXML files
- `QScxmlStateMachine` provides functions
  - to access states
  - to observe state changes and events
  - to send events
  - to access invoked services to access data models
- `QScxmlDataModel` sub-classes allow accessing data model locations and expressions

Contents

Objectives

Learn…

- …how to launch and terminate processes
- …how to communicate between processes with standard input and output
- …IPC options in Qt
- …how to use shared memory
- …how to use Desktop-Bus
- …how to observe changes in the file system

Processes

- `QProcess` allows launching external programs and communicating with them
  - Both synchronously and asynchronously
- After the process has been created, it enters the `Starting` state
  - After the process is started, it enters Running state and emits `started()` signal
- Process may be started several times (platform dependent behavior)

```
QString program = "./helloworld";
QStringList arguments;
arguments << "-style" << "motif";

QProcess *aProcess = new QProcess(parentObject);
aProcess->setProgram(program);
aProcess->setArguments(arguments);
aProcess->start();
```

Asynchronous Process Invocation

- `QProcess::start()` starts the process asynchronously
  - Function returns possibly before the child process is running
  - Signals `started()` or `errorOccurred()` tell whether the process was started successfully
  - `start()` may be called several times – no effect on the running process

- Static `QProcess::startDetached()` starts a child process and detaches it from the current one
  - This method will not wait for termination, and the child process will not be terminated when the current process terminates ("fire and forget")

Synchronous Process Invocation

- Wait until the child process has started (or finished)
  - `waitForStarted()` returns when the `started()` signal has been emitted
  - `waitForFinished()` returns when the `finished()` signal has been emitted

- `execute()` Is another way to start a process synchronously
  - Starts a process and waits for its termination
  - Does not allow processing the child input or sending output to the child

```
QStringList arguments;
arguments << "Argument1" << "Argument2";
QProcess::execute("do_it_now", arguments);
// won't get here until do_it_now terminates
```

- Calling these methods in the main (GUI) thread will freeze your user interface

Inter-Process Communication

- As straightforward as accessing the files
  - Thanks to `QIODevice`
- Write process's standard input using `write()` and read from the standard output using `read()`, `readLine()`, `readAll()`, `getChar()`

```
QProcess gzip;
gzip.start("gzip", QStringList() << "-c");
if (!gzip.waitForStarted())
    return false;
gzip.write("Hello World!");
gzip.closeWriteChannel();
if (!gzip.waitForFinished())
    return false;
QByteArray result = gzip.readAll();
```

Inter-Process Communication Options

- `QSharedMemory`
  - Reference count object, can be opened by any process
  - Simple `memcpy()` used to read/write to the process
- Servers (`QCoreApplication` instances)
  - `QLocalSocket` used (local loop TCP socket)
- DBus
  - Extends signal/slot mechanism between processes
  - DBus protocol must be supported by the platform
- QCop (Qt Communication protocol)
  - Available only in Qt for embedded Linux prior Qt 5
- Platform-dependent functionality
  - Message queues, pipes

Shared Memory

- `QSharedMemory` class
- Works between processes and threads
  - Processes recognize the piece of shared memory using a key (`QString`)
  - Process attach and detach to shared memory using the key
- Do not de-allocate shared memory buffer
  - Reference count – will be freed, when all `QSharedMemory` objects referencing it have been deleted
- Mutual exclusion is taken care by the developer
  - Use `lock()` and `unlock()`

Shared Memory Example

```
QSharedMemory mem(QString("thekey"), 0);

// Create a new/old shared memory
mem.create(SIZE);
mem.lock(); // uses QSystemSemaphore internally
char *to = (char*)mem.data();
const char *from = buffer.data();
memcpy(to, from, qMin(mem.size(), SIZE));
mem.unlock();
```

D-Bus

- D-Bus itself is a server
  - Message (remote procedure call) –based communication between processes
- Applications (service providers) send messages to D-Bus, which routes the messages to one or more receiver
- Abstract
  - D-Bus does not define, which mechanism (e.g. sockets) actually transfers the messages
- Two kinds of daemons
  - System-wide singleton (for system messages, such as signal strength, battery level)
  - User session –specific (between applications)

D-Bus Concepts

- Object **paths**
  - A mechanism to locate, which native object (GObject, Java Object, Qt QObject) provides a service
  - E.g. `company/services/serviceX`
- Each object may have method and signal **members**
  - Methods are (remote procedures) operations which can be invoked on an object with optional input and (possibly several) output values
  - Signals are broadcast from an object to all its observers (may contain data)
  - E.g. `doSomething, notify`
- Member group is mapped to an **interface**
  - Mapped to Java interface or C++ pure virtual class
  - Identified as `com.company.InterfaceName`

D-Bus Concepts

- **Bus names**
  - D-Bus daemon assigns a unique connection name for each connection from applications
  - After a name is mapped to an application, the application owns that name
  - Applications may ask to own well-known names, e.g. `com.theqtcompany.MessageEditor`
- **Addresses**
  - Specify where a server will listen and where a client will connect
  - Possibly, your service is a server daemon to which applications send messages

Qt DBus

- Allows to call methods of D-Bus objects
- Allows to connect signals and slots between D-Bus objects
- Since it uses the meta object information, it is not necessary to know the interface of the remote object
- Takes care of mapping Qt data types to the defined D-Bus data types
- Resolves object names to interfaces with the correct signals and slots

Calling Methods on D-Bus Objects
Client Side

- In Qt DBus, the slots on the remote object can be called as if the object was local
- To call a method on the remote object, `QDBusInterface` has to be retrieved for it first
- The method can then be called using `QDBusInterface::call` or `QDBusInterface::asyncCall()`

```
QDBusReply<QString> reply = iface.call("echo", "hi");
```

  - calls the slot named echo on the remote object with argument "hi"

- The interface object may be created from D-Bus XML interface using **qdbusxml2cpp –p** tool
  - Generates public slots and signals in the interface class
  - May be directly accessed from the client
  - Called, if the variable `DBUS_INTERFACES += interface.xml` defined

Mapping between QtDBus and D-Bus Data Types

- Qt DBus needs to map Qt data types to types known by D-Bus
- All arguments marshalling is taken care of by Qt
- Supported data types: `uchar`, `bool`, `short`, `ushort`, `int`, `uint`, `qlonglong`, `qulonglong`, `double`, `QString`, `QStringList`, `QByteArray`, and special D-Bus types
- Compound types can be formed as arrays, structs, and maps
- To use custom data types,
  - declare the type using `Q_DECLARE_METATYPE()`,
  - and register it using `qDBusRegisterMetaType()`

Providing Methods on D-Bus Objects
Server Side

- `QDBusConnection::sessionBus()/systemBus()`: access to the bus objects
- `QDBusConnection::registerService()`: register a service (“host part”)
- `QDBusConnection::registerObject()`: register an object (“file part”)
- `QDBusInterface` constructor constructs a `QObject` that represents the signals and slots of the remote object

Qt D-Bus Server Implementation

- Create a service object
  - Often created with **qdbusxml2cpp –a** tool
    - `qdbusxml2cpp –a myAdaptor.h: myInterace.xml`
    - `qdbusxml2cpp –i myAdaptor.h –a :myAdaptor.cpp myInterace.xml`
  - Maps D-Bus messages to signals and slots
    - `new serviceAdaptor(myServerObject);`

- Create a D-Bus session

```
if (!QDBusConnection::sessionBus().isConnected()) { /* Handle error */ }
if (!QDBusConnection::sessionBus().registerService(SERVICE_NAME)) { }
```

Qt D-Bus Server Implementation

- Register the service
  - `QDBusConnection::sessionBus().registerObject("/", &myServerObject, QDBusConnection::ExportAllSlots);`
- The exposed signals and slots can be restricted by the remote object (`QDBusConnection::RegisterOptions()`)

File Watcher - QFileSystemWatcher

- Monitors file and directory changes
- Several files and directories can be monitored at the same time
  - Platform may set limitations on the number of monitored files
- Provides signals `fileChanged()`, `directoryChanged()` to notify, which file or directory path changed

```
m_watcher->addPath(QDir::tempPath());
connect(m_watcher, &QFileSystemWatcher::directoryChanged,
        this, &FileLoader::doSomethingWithNewOrChangedFiles);
```

Questions and Answers

- How processes can be started in Qt?
- When would you wait for a process to be started or finished? When should you not synchronously wait for the start?
- How Qt uses process standard input and output?
- What inter-process communication options exists in Qt?
- What are good use cases for using shared memory? When would you use D-Bus, provided it is available in your platform?

Summary

- `QProcess` supports starting and terminating processes as well as communicating between processes
- It is useful to start processes synchronously, if your thread needs the data from the other process before proceeding
- `QProcess` derives from `QIODevice`, which provides an API for inter-process communication
- Shared memory, files, Desktop-Bus, and platform-specific pipes and message queues are other options for inter-process communication
- Shared memory is useful for server solutions, where the server creates and manages the shared memory
  - Clients read/write to the shared memory
- D-Bus provides a signal/slot-based inter-process communication

Contents

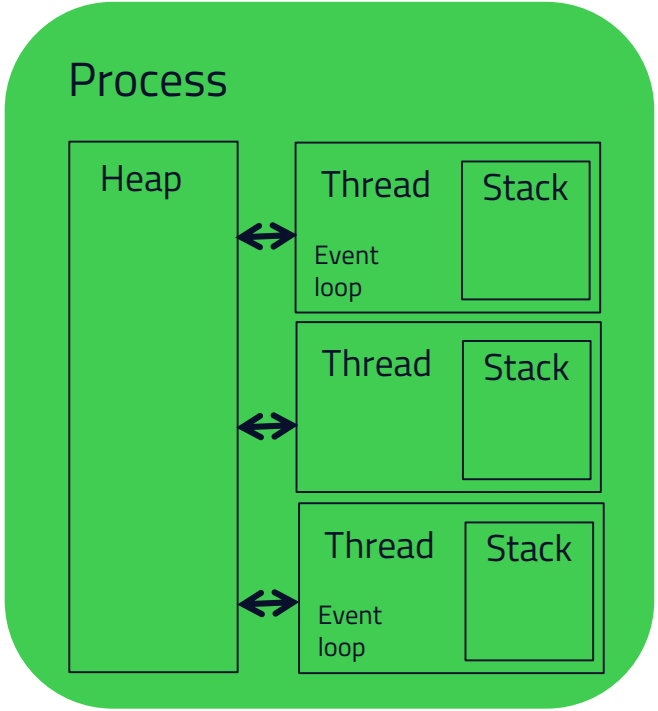
Objectives

Learn…

- …Qt threading options
- …how to use QThread correctly
- …how to properly communicate between Qt objects in different threads
- …how to manage the thread life time
- …how to use QRunnable and thread pool

Qt Threading Model

- Qt uses platform threads, managed via `QThread`
  - No Qt-specific threads or scheduler
- By default one process has one thread
  - Main thread or GUI thread in GUI apps
- Each thread share a common heap, but has its own stack
- `QThread` has a thread-specific event loop
  - It is not necessarily running



Threading Options

	Use Cases
QThread	- Developer wants to manage the thread life time (create, start, finish) - There is only a single task or several different tasks, needed to be executed concurrently
QtConcurrent	- High-level multithreading - Threads are re-cycled by the thread pool - Item container manipulation concurrently - There are several similar tasks, which needs to be executed concurrently - Tasks may return a value
QRunnable	- Low-level multithreading - Threads are re-cycled by the thread pool - Several threads with similar functionality needed - Tasks return void

Reentrant Classes

- All member functions are re-entrant
- A class may be used in multiple threads, but each thread has its own instance of the class
- Many classes are re-entrant among 1,500 classes in Qt libraries
- Most implicitly shared value types
  - Not `QPixmap`
- Many `QObjects`, no widgets though
  - `QSvgGenerator` and `QSvgRenderer`
  - Rich text processing classes, like `QTextDocument` with even `clone()` function
- You may need to explicitly create a copy of a re-entrant object for another thread

Thread-Safe Classes

- All member functions are thread-safe
- The class instance may be shared by multiple threads – mutual exclusion needed
- Very few Qt classes are thread-safe – why?
  - Mutex, semapahore, wait condition
- Some functions are thread-safe
  - `QObject::connect()`
  - `QCoreApplication::postEvent()`
  - Signal emission

Thread Affinity

- Each Qt object belongs to zero or one thread
  - By default the thread, in which the object is created
- Creating a Qt object in one thread and calling its functions from another thread is not guaranteed to work
  - You must not interrupt object in the middle of event handling by calling its functions from another thread
  - You must not delete an object from another thread, if the object is still handling events
  - You must not access widgets from other than the GUI thread
- Event-based classes must be used in one thread
  - You cannot create and start a `QTimer` in two separate threads
  - You cannot create and use a `QTcpSocket` in two separate threads

Thread Affinity Solutions

- Thread affinity may be changed
  - Qt object must be reentrant
  - Qt object cannot have a parent
- `QObject::moveToThread(QThread *target)`
  - Pushes an object to another thread – no way to pull an object from the thread
  - Qt object member pointers move only, if their parent is moved as well
- For inter-thread communication
  - Use signals with auto connection type, if the thread affinity can be changed
  - Use posted events

QThread

- Think of `QThread` as a manager object
  - Priority, thread execution, stack size
- Constructor is executed in the caller thread
  - Each Qt object created in the constructor belong to the creator thread

```
Thread::Thread(QObject *parent) :
    QThread(parent),
    m_memberPointerToQObject(this) // To change the thread affinity
{
    setObjectName("Child thread");
    qDebug() << "Current thread" << QThread::currentThread();

    // Timer thread is changed to this thread
    m_timer.moveToThread(this);
}
```

Thread Programming

- To create a new thread, instantiate `QThread`
  - Sub-classing is is possible, but not recommended
- Create a worker object or objects
  - Derive from `QObject`
  - Define signals/slots needed to communicate safely with the object
  - Move the worker's affinity to the new thread
- Possibly your thread does not have any worker objects
  - Then just sub-class `QThread` and re-implement the `run()` method
- Set the priority, stack size, if needed
  - `IdlePriority, …, TimeCriticalPriority`
  - Priority may be "inherited" from the parent thread
- Start the thread by calling `start()`

Creating a Thread with a Worker

```
QThread *thread = new QThread();
Worker *worker = new Worker();

connect(worker, &Worker::error, errorHandler, &ErrorHander::errorString);
connect(thread, &QThread::started, worker, &Worker::process);
worker->moveToThread(thread);

// Worker knows when it is finished
connect(worker, &Worker::finished, thread, &QThread::quit);

connect(worker, &Worker::finished, worker, &Worker::deleteLater);
connect(thread, &QThread::finished, thread, &QThread::deleteLater);
thread->start();
```

Running a Thread

- Default implementation of `QThread::run()` does nothing else but calls `exec()` to start the event loop
  - Re-implement, if no event loop needed
- Event loop is needed for handling events
  - Queued connections are based on events
- Functions to check thread state
  - `QThread::isFinished()`, `QThread::isRunning()`
- Functions to temporarily stop thread execution
  - `QThread::sleep()`, `QThread::msleep()`, `QThread::usleep()`
  - `QTimer` should be preferred to enable event handling in the thread

Queued Connections and Signal Arguments

- Serialize signal arguments into an event object, posts the event, handles the event, re-creates the argument objects in the receiver thread using object introspection, and calls the slot
- Pass a value-type as copy

```
SIGNAL(someSignal(CustomType)) // Copies the arg, before sending an event
```

- Pass a value type as reference

```
SIGNAL(someSignal(const CustomType &)) // Argument is copied
```

- Pass a Qt object type

```
SIGNAL(someSignal(CustomType *)) // Pointer is copied, mutual exclusion may be
needed
```

- Pass a shared object, which may be deleted by any thread at any time

```
SIGNAL(someSignal(QSharedPointer<CustomType>)) // Mutual exclusion may be needed
```

Graceful Thread Cleanup

- Avoid terminating a thread
  - Risk that allocated resources in a shared heap are not cleaned up
- If a thread has an event loop
  - Quit the event loop – `QThread::quit()`
- Thread may be stopped from another thread
  - `QThread::requestInterruption()`
  - Check periodically `QThread::inInterruptionRequested()`
  - No event loop needed
- If your threads runs a busy loop
  - No event handled – no timer events
  - Call `QThread::eventDispatcher()->processEvents()` periodically

Graceful Thread Cleanup

- Just right before thread finishes its execution, it emits `finished()` signal
  - Thread has quite the event loop
  - No more events can be handled
  - Deferred deletions are still executed
- Useful to delete allocated thread resources
  - Use `deleteLater()` to delete the worker and thread objects

Mutual Exclusion

- Mutexes are implemented by the class `QMutex`
- The two important methods are `lock()` and `unlock()`
- You can try locking a mutex using `tryLock()` or `try_lock() // std-compatible`
  - If the lock was obtained it will return true, otherwise it will return false right away, rather than waiting for the mutex
- `tryLock(int timeout)` or `try_loc_for()` will wait timeout milliseconds before giving up on getting the lock

Thread Synchronization

- `QMutex`
  - Protects access to a shared resource
  - Recursive locking supported
- `QReadWriteLock`
  - Increases concurrency compared to `QMutex`
  - Multiple reads allowed
- `QSemaphore QSystemSemaphore`
  - Protects a certain number of identical resources
- `QWaitCondition`
  - Several threads may wait for a condition
  - It is possible to wake up one thread randomly or all the threads
  - One thread waits, another thread wakes it up

Hint! The system semaphore is a kernel object, but other locks are simple counters protected with atomic operations. So use a system semaphore only, if you need to synchronize threads running in separate processes

QMutexLocker

- When you lock a mutex you must, of course, unlock it again!
- This can be troublesome if you want to lock a mutex at the entrance of a function, and unlock it at exit—your function can possibly return from many places (code like “if (...) return false;”)
- If you are using exceptions (or libraries that do), every statement can be an exit point from your function!
- `QMutexLocker` will help you here, simply put the following code right before you need the lock, and it will lock the mutex for the duration of the block:
  - `QMutexLocker lock(&myMutex);`

QMutexLocker

```
QMutex sharedMutex;

class Simple
{
public:
    Simple() { n = 0; }

    void increment() { QMutexLocker locker(&sharedMutex); n+=2; }
    void decrement() { QMutexLocker locker(&sharedMutex); n-=2; }
    int value() const { QMutexLocker locker(&sharedMutex); return n; }

private:
    int n;
};
```

Wait Condition

- `QWaitCondition::wait()` lets a thread wait for a certain event
  - You can specify a maximum waiting time
- You must pass a locked `QMutex` (no `QReadWriteLock`, though), to atomically go from locked state to wait state
  - The mutex will be automatically locked before the thread is woken
- Wake one (random) thread waiting on a wait condition with `QWaitCondition::wakeOne()` and all waiting threads with `QWaitCondition::wakeAll()`

```
Q_FOREVER{
    mutex.lock();
    keyPressed.wait(&mutex);
    do_something();
    mutex.unlock();
}
```

QRunnable Interface

- `QRunnable` can be used instead of `QThread`

- Light-weight way of implementing multithreading
  - No need to manually create/delete a new thread object – threads are re-cycled
  - A free thread is picked from `QThreadPool`
  - If no free thread exists, the task is queued

```
class HelloWorldTask : public QRunnable
{
// Note: QRunnable does not have a base class!
    void run() {
        qDebug() << "Hello world from thread" << QThread::currentThread();
    }
}
HelloWorldTask *hello = new HelloWorldTask();
// QThreadPool takes ownership and deletes 'hello' automatically
QThreadPool::globalInstance()->start(hello);
```

QThread versus QRunnable

- `QThread` derives from `QObject`
  - Signals and slots
  - `QObject` is “heavy”
  - Cost of creating a thread
- `QRunnable` has no base class
  - Light-weight
  - Runs on any free thread
  - Designed to be used `QThreadPool`
  - By default deleted by the thread pool

Thread Pool

- Manages threads in the global thread pool in the application
- Possible to set max thread number – by default `QThread::idealThreadCount()`
  - Releases threads, if threads are idle for a defined time period – by default 30s
- Possible to `clear()` the queue or `cancel()` one or more tasks

```
QThreadPool *threadPool = new QThreadPool;
// threadPool->waitForDone();
…
void somewhereElse()
{
    MyTask *task = new MyTask;
    threadPool->start(task); // also tryStart() function
}
```

Thread Reservation

- A thread may be reserved for handling blocking functionality
  - There is always at least one thread in the pool, even tough max thread count < 0
- If active thread count equals to max thread count and the thread blocks, waiting a new child thread to complete, there is a deadlock
  - Solution is to temporarily increase the thread count beyond maximum by calling `QThreadPool::reserveThread()`
  - After blocking functionality a thread is released with `releaseThread();`
- It is also possible to yield execution of a thread to other threads by releasing a thread before reserving it
  - The thread will wait `reserveThread()` to be called
- `QThread` has also a static `yieldCurrentThread()` function

Questions and Answers

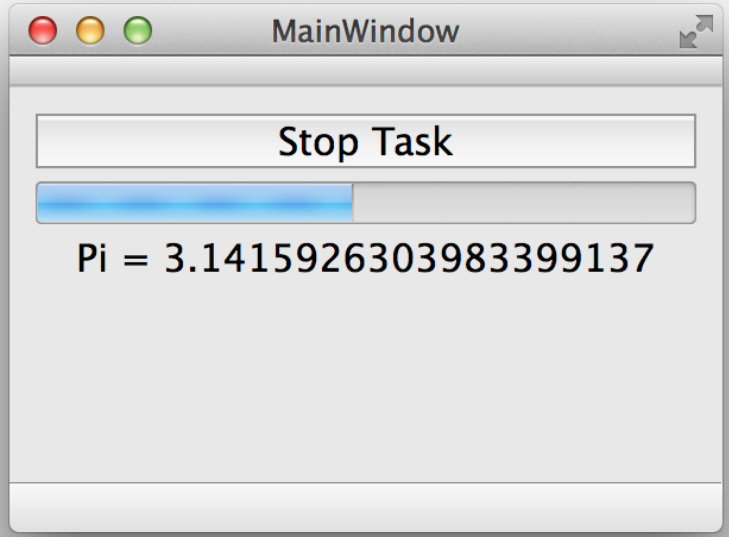
- What multithread options are there in Qt?
- When would you use `QThread` and when either low-level or high-level multithreading API?
- When do you need an event loop in a thread?
- Do threads, created by the `QThreadPool`, have an event loop?
- Why is it often recommended not to subclass `QThread`?
- Why is it important to make sure Qt object has the right affinity?
- How can you notify from `QRunnable` that the task has finished?
- How many threads are available in `QThreadPool`?
- How many threads can be running in a Qt program?
- Why should not you kill or terminate a thread?

Summary

- `QThread` is a Java-like API to multithreading
  - `QThread::start()` will result `QThread::run()` to be called in a child thread
- Thread affinity defines to which thread a Qt object belongs to
  - Can be NULL, in which case signal/slot, event handling, event filters do not work
  - In many cases, developer should take care the QT object members are not called outside the thread affinity
  - Qt objects may be moved to other threads to guarantee the correct thread affinity
  - Posted events allow calling Qt object members from another tread safely
- `QRunnable` interface is similar to Java Runnable
  - Qt runnable objects re-use threads using `QThreadPool`
  - No performance penalty of creating and deleting threads

Lab – Pi Calculator

- You are provided with a worker object, which calculates pi digits
- Your task is to run the worker in its own thread
- Pay attention to
  - Proper memory management
  - Proper thread termination and cleanup
  - Communication between the worker and UI widgets
- Read the implementation details in readme.txt



Contents

Objectives

Learn…

› …how to use QtConcurrent name space to run concurrent tasks

› …how to synchronize tasks

› …how to manipulate item containers concurrently

Qt Concurrent

- High-level framework for parallel algorithms
  - Actually a namespace
- Work is automatically distributed over an optimal number of threads, determined at runtime
  - Based on the thread pool like `QRunnable` interface
- Supports executing concurrent tasks `QtConcurrent::run()`
- Supports manipulating item containers concurrently

Concurrent Tasks

- `QFuture<T> QtConcurrent::run(Function function,…)`
- `QFuture` is a result of an asynchronous computation
  - Not `QObject`
  - Provides pause and resume functionality
  - Provides progress information
  - Allows functionality to iterate through the results
  - Other useful functions:
    - `isFinished()`
    - `isRunning()`
    - `isStarted()`
    - `waitForFinished()`
- Uses a free thread from a thread pool

QFuture Example

```
int myFunction()
{
    int result(0);
    Q_FOREVER {
        // Calculate result
        if (thread()->isInterruptionRequested())
            return result;
    }
}

void somewhereElse()
{
    QFuture<int> result = QtConcurrent::run(myFunction);

    // do some other work in parallel to myFunction
    result.waitForFinished();
}
```

Other QFuture Functions

- Multiple `QFuture` can be combined in a `QFutureSynchronizer`
- For non-blocking synchronization, there is `QFutureWatcher`
  - Uses signals and slots
  - Enables the event driven functionality with threads

Future Examples

```
QFuture<int> f1 = ...; // QtConcurrent::run(…)
QFuture<int> f2 = ...;

QFutureSynchronizer<int> sync;
sync.addFuture(f1);
sync.addFuture(f2);
sync.waitForFinished(); // blocks

QFuture<int> future = ...;
QFutureWatcher<int> watcher;
watcher.setFuture(future);
connect(&watcher, &QFutureWatcher::finished(), this, &Observer::slotFinished);
```

Concurrent Container Manipulation

- Data in a container may be transformed or filtered

- Manipulate data in-place - `map()`, `filter()`)
- Copy data into a new container - `mapped()`, `filtered()`)
- Optionally, use reduction - `mappedReduced()`, `filteredReduced()`)
- Blocking mapping and filtering functions exist as well - `blockingMapped()` returns the result or asynchronous `mapped()` returns a future

- The algorithms are defined in namespace `QtConcurrent`

QtConcurrent — Mapping

- `QtConcurrent` can transform (map) sequences based on a user-defined mapping function
- Only random access sequences (`QVector`, `QList`) should be used with `QtConcurrent`, forward sequences (`QLinkedList`, `QMap`,. . .) can be used, but incur a performance penalty
- The mapping function either takes one element of the sequence as an argument and returns the modified element (`mapped()`), or modifies the argument directly (`map()`)
- The order in which elements are processed is undefined, though the sequence is never reordered

QtConcurrent — Filtering

- `QtConcurrent` can filter (grep) sequences based on a user-defined filter function
- The filter function takes one element of the sequence as an argument and returns true (keep element) or false (drop element)
  - Filter functions are “unary predicates”
- Filter and mapping functions may also be member functions of the elements in the sequence

```
QStringList input = ...;
QStringList lower =
QtConcurrent::blockingMapped(input, &QString::toLower);
```

- Filtering and mapping are very similar, so in the following, we talk about mapping, and point out where filtering differs

QtConcurrent — Reduce Operation

- In addition to mapping/filtering, `QtConcurrent` can optionally reduce the sequence with a user-defined reduce function

- The reduce function takes the partial result by reference, and the next element of the sequence as arguments and modifies the partial result to incorporate the new element
  - The return value is ignored

```
void join(QString &result, const QString &next) {
    result += next;
}
```

- `QtConcurrent::ReduceOptions` specify how exactly the reduction is applied

- Currently, reduction is never parallelized
  - The mapping part is parallelized

Questions and Answers

- What is Qt Concurrent?
- What are the differences between `QtConcurrent::run()` and `QRunnable::run()`?
- How container data can be manipulated?
- Is it possible to use Qt Concurrent in single core CPUs?
- What should be taken into account in terms of containers, when using Qt Concurrent?

Summary

- Qt Concurrent provides a high-level API for multitasking
- Compared to low-level API, there is no for sub-classing
- Tasks may also return values, wrapped into `QFuture` objects
- Item containers may be transformed and filtered concurrently
  - Useful for random access sequences as the processing order is undefined

Contents

- TCP/UDP Sockets
- WebSockets
- SSL Sockets
- QNetworkAccessManager
- Requests and Replies
- DNS and Proxies
- Cookies

Objectives

Learn…

› …how to use TCP sockets, SSL sockets, and web sockets

› …how to make network requests and handle network replies

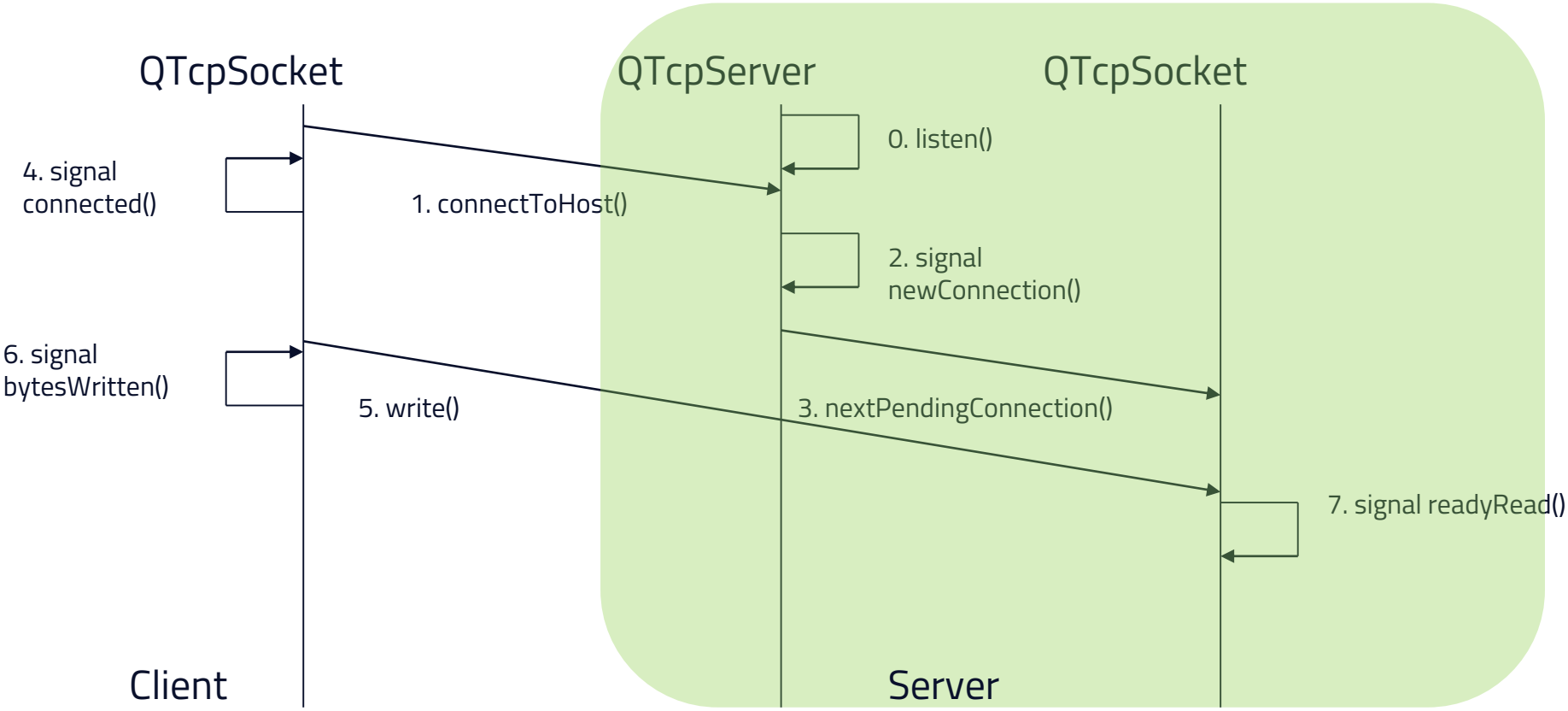
Qt Network

- Easy to use with high-level classes
  - Instead of `QHttp` and `QFtp`, use `QNetworkRequest`, `QNetworkAccessManager`, `QNetworkReply`
  - `QTcpServer`
  - `QTcpSocket`
  - `QUdpSocket`
  - `QHostInfo`
  - `QNetworkInterface`
  - `QNetworkProxy`
- Add network module to your project file
  - QT += network

Sockets

- UDP sockets
  - Write is automatically flushed and a signal is emitted
  - Read event is handled by the event loop and `readyRead()` signal is emitted
  - Check the availability of data using `bytesAvailable()`
  - Read the data
- TCP connections handled in the same way
  - Set up the server by calling `listen()`
  - Connect to `newConnection()` signal
  - In the slot, call `nextPendingConnection()`, which returns a `QTcpSocket` object to communicate with the client
- Socket returned by `nextPendingCnnection()` cannot be used in another thread
  - Sub-class `QTcpServer` and re-implement `incomingConnection()` to get a socket descriptor
  - Use the descriptor in another thread

Socket Sequence Diagram



TCP Client Implementation

- Create an instance of `QTcpSocket`
- Call `QAbstractSocket::connectToHost()`
- Data can be read and written using `QIODevice read()` and `write()` functions
- Data may be serialized to the socket using `QDataStream`

```
QDataStream out(tcpSocket);
out << serializedObject;
```

  - Note that Qt has several versions of `QDataStream`
    - Use `setVersion()` if necessary

TCP Client Implementation

- The signal `readyRead()` is emitted whenever data is avalable to be read on the socket
  - `QDataStream in(tcpSocket);`
  - `in >> size;`
  - `in >> string;`

- Data may be sent in fragments
  - Not possible to de-serialize a large image directly from the stream, for example
  - The result would be incomplete image objects
  - Both `QDataStream` and `QIODevice` support transactions

```
QDataStream in(m_socket);
in.startTransaction();
QByteArray possiblyLargeDataArray;
qint32 someValue;
in >> possiblyLargeDataArray >> someValue;
if (!in.commitTransaction()) return;
createImage(possiblyLargeDataArray); // Now all data has been completely written
```

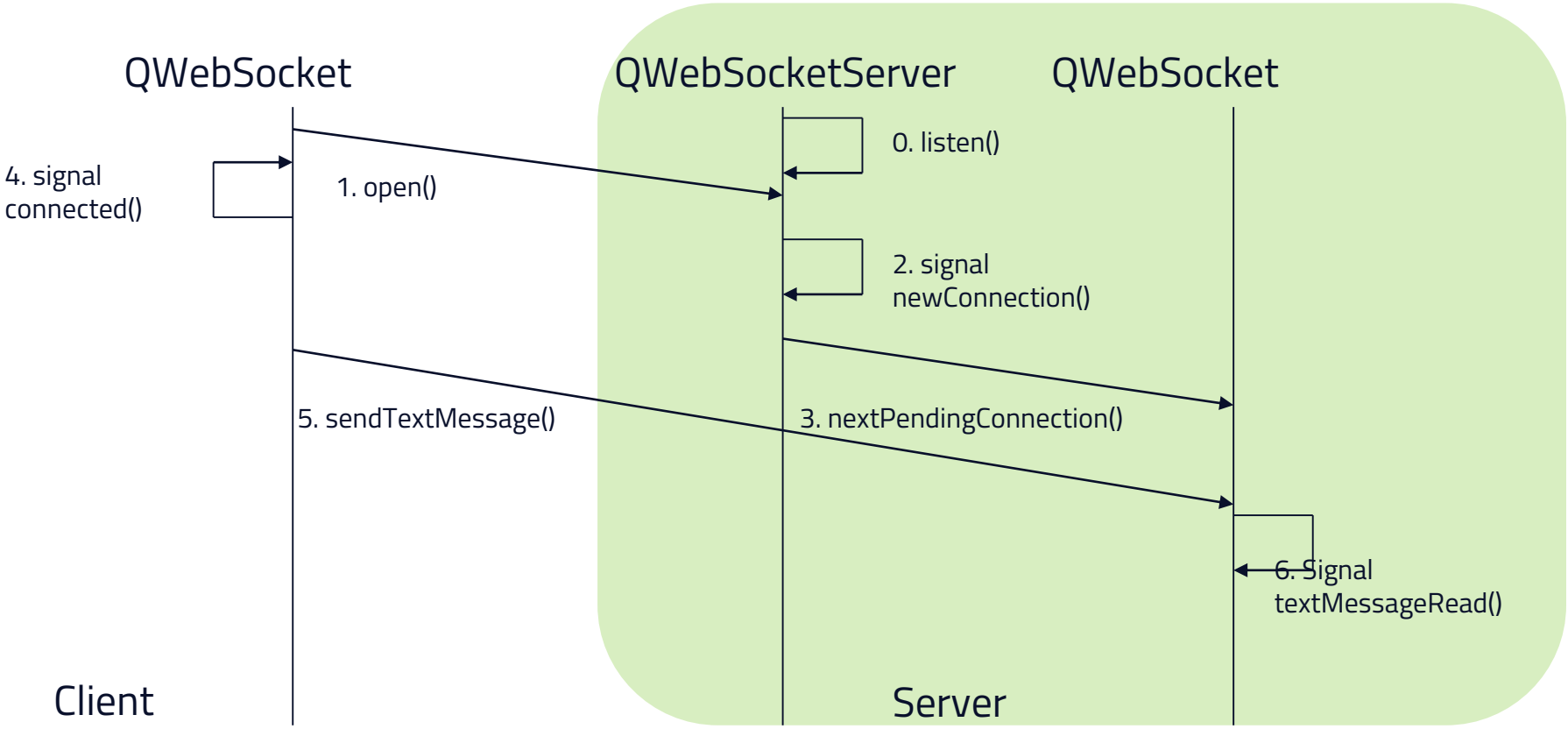
TCP Server Implementation

- Create an object of class `QTcpServer`
- Call `listen()` on that object
- You can either specify the port to listen to or let `QTcpServer` pick a free one
  - `serverPort()` will tell you the one it is using
- When a connection is made, the `newConnection()` signal is emitted
- Upon this, call `nextPendingConnection()` to get a `QTcpSocket` that is already connected to the client, and that you can then use for communication

WebSockets

- TCP connection, where the communication takes place using Web Socket protocol (`ws://host:port`)
- Like in `QNetworkAccessManager`, it is possible to set an SSL configuration and a proxy
- API very similar to TCP
  - `QWebSocketServer` – `QTcpServer`
  - Listens for connections, establishes connection
  - `QWebSocket` – `QTcpSocket`
  - Requests for a connection
  - Transfers data

WebSocket Sequence Diagram



QSslSocket

- The class `QSslSocket` supports secure network access using either the SSLv3 protocol or the TLSv1 (default) protocol
- `QSslSocket` inherits from `QTcpSocket`, and, after setup, the communication is just like with a `QTcpSocket`
- Only supported backend for SSL is OpenSSL, which needs to be installed separately
  - Can be installed after the configuration

Ssl Socket Clients

- The common way for clients is to call `QSslSocket::connectToHostEncrypted()`, which is similar to `QTcpSocket::connectToHost()`, except that it will set up a secure connection

- After the connection request, clients should either call `waitForEncrypted()` or connect to the `encrypted()` signal
  - The signal is emitted after the secure connection has been established
  - Data may be written to the socket immediately after `connectToHostEncrypted` call (data will be queued)

```
QSslSocket *socket = new QSslSocket(this);
connect(socket, SIGNAL(encrypted()), this, SLOT(ready()));
socket->connectToHostEncrypted("address.com", 993);
```

Ssl Socket Servers

- The easiest way to implement a SSL server is to inherit from `QTcpServer`, and override `incomingConnection(int socketDescriptor)`
- A `QSslSocket` is then constructed based on the socket descriptor
- Once this is set up, handshaking is started using `startServerEncryption()`

```
void SslServer::incomingConnection(int socketDescriptor)
{
    serverSocket = new QSslSocket;
    if (serverSocket->setSocketDescriptor(socketDescriptor)) {
        connect(serverSocket, SIGNAL(encrypted()), this,
                              SLOT(ready()));
        serverSocket->startServerEncryption();
    }
}
```

Network Access Manager

- Instead of direct HTTP protocol interface (`QHttp`) it is recommended to use `QNetworkAccessManager` interface
  - Create a `QNetworkAccessManager` object
  - Call a desired function (get(), post(), head(), post()) with one `QNetworkRequest` holding the URL
  - Receive a `QNetworkReply` object as the response
  - In addition `setProxy()` and `setCookieJar()` configure proxies and cookie handling

```
QNetWorkAccessManager* accessManager = new QNetworkAccessManager(this);

accessManager->get(new QNetworkRequest(new
                        QUrl("http://lively.cs.tut.fi/images/add.ico")));

connect(accessManager, SIGNAL(finished(QNetworkReply*)), this,
                        SLOT(doStuffWithResult(QNetworkReply*)));
```

Network Request

- The argument to the methods of `QNetworkAccessManager` are instances of `QNetworkRequest`
  - Requests are queued by the network access manager
  - Requests are handled in parallel (6 on the desktop platforms)
- In the simplest setup, `QNetworkRequest` is created with a `QUrl` as argument
- SSL is configured using `setSSLConfiguration()`
  - No default configuration – selected by the backend
- Raw headers may be configured using:
  - `setHeader(KnownHeaders headerName, QVariant headerValue)`
  - `setRawHeader(QByteArray headerName, QByteArray headerValue)`

Network Reply

- The methods of `QNetworkAccessManager` are all asynchronous
- The result of the calls are instances of a `QNetworkReply`
- The signals `QNetworkReply::finished()` and `QNetworkAccessManager::finished(QNetworkReply*)` tells you when the operation is done
- The signal `downloadProgress(qint64, qint64)` respectively `uploadProgress()` informs you about progress

Network Reply

- Errors are signaled with `error(NetworkError)` - a printable string may be obtained from `errorString()`
- `QNetworkReply` is a subclass of `QIODevice`
  - Uses sequential (rather than random) access
- Note: : It is your responsibility to delete the `QNetworkReply` object
  - Do not delete in the slot (use `deleteLater()`)

ex-multi-download

Additional Features

- Network configurations (`QNetworkConfiguration`)
  - May be set explicitly by the developer
  - Use WiFi, CDMA, 4G, Ethernet, …
- Cache (`QAbstractNetworkCache`, `QNetworkDiskCache`)
  - Allows storing data into any `QIODevice` using streaming operators
  - Maximum size and cache load control may be set
- Cookies (`QNetworkCookieJar`, `QNetworkCookie`)
  - Name, value, secure, domain (**./foo/bar**)
- Proxy (`QNetworkProxy`)
  - Type, host name, port, user, password

Authentication

- Whenever authentication is required, a signal `QNetworkAccessManager::authenticationRequired(QNetworkReply, QAuthenticator)` emitted
  - Direct connection must be used (authentication credentials must be provided when the signal returns)
  - Credentials cached by the network access manager
- Read the header information from the reply
- Set user name and password in the authenticator

Proxies

- Proxies can be set up with the class `QNetworkProxy`
- `QNetworkProxy` is used to identify HTTP, FTP and SOCKS5 proxies
- HTTP and FTP proxies can perform caching
- To use a proxy:
  - Create a `QNetworkProxy` object and populate it with hostname, port, etc.
- Assign the proxy globally with the static method `QNetworkProxy::setApplicationProxy()` or just on one socket using `setProxy()`

Customizing Proxies

Proxy factories are used to create policies for proxy use

- `QNetworkProxyFactory` supplies proxies based on queries for specific proxy types
- Queries are encoded in `QNetworkProxyQuery` objects
- `proxyForQuery()` is used to query the factory directly
- To change the behavior, reimplement `queryProxy()`
- To implement an application-wide policy with the factory, call `setApplicationProxyFactory()`
  - This overrides any proxy set with `QNetworkProxy::setApplicationProxy()`
  - Querying `QNetworkProxy::applicationProxy()` causes the factory to be queried

Proxy Queries

Queries enable proxies to be selected based on key criteria:

- The purpose of the proxy: TCP, UDP, TCP server, URL request
- Local port, remote host and port
- The protocol in use: such as HTTP or FTP
- The URL being requested

Questions and Answers

- How web sockets API differs from TCP socket API?
- What options exist for reading/writing data using sockets?
- What should be taken when using sockets in a multi-threaded program?
- How can you make REST API requests?
- Would it make sense to handle network access manager requests in separate threads to keep the GUI thread responsive?
- Is there anything in common between a TCP socket and network reply?
- How cookies are managed in Qt?

Summary

- Qt network module provides several classes for networking
  - UDP socket classes
  - TCP sockets
  - SSL sockets
  - Host name resolving services
- For HTTP and FTP networking, use `QNetworkAccessManager`
  - `QHttp` and `QFtp` still work in Qt5 as an add-on module, so your old programs do not need re-implementation
- Network access manager provides classes for making any kind of a network request and handling any kind of a network reply

Contents

Objectives

Learn…

› …the overall class hierarchy of Qt WebEngine classes

› …asynchronous nature of some WebEngine APIs

› …how to use Qt objects APIs in WebEngine

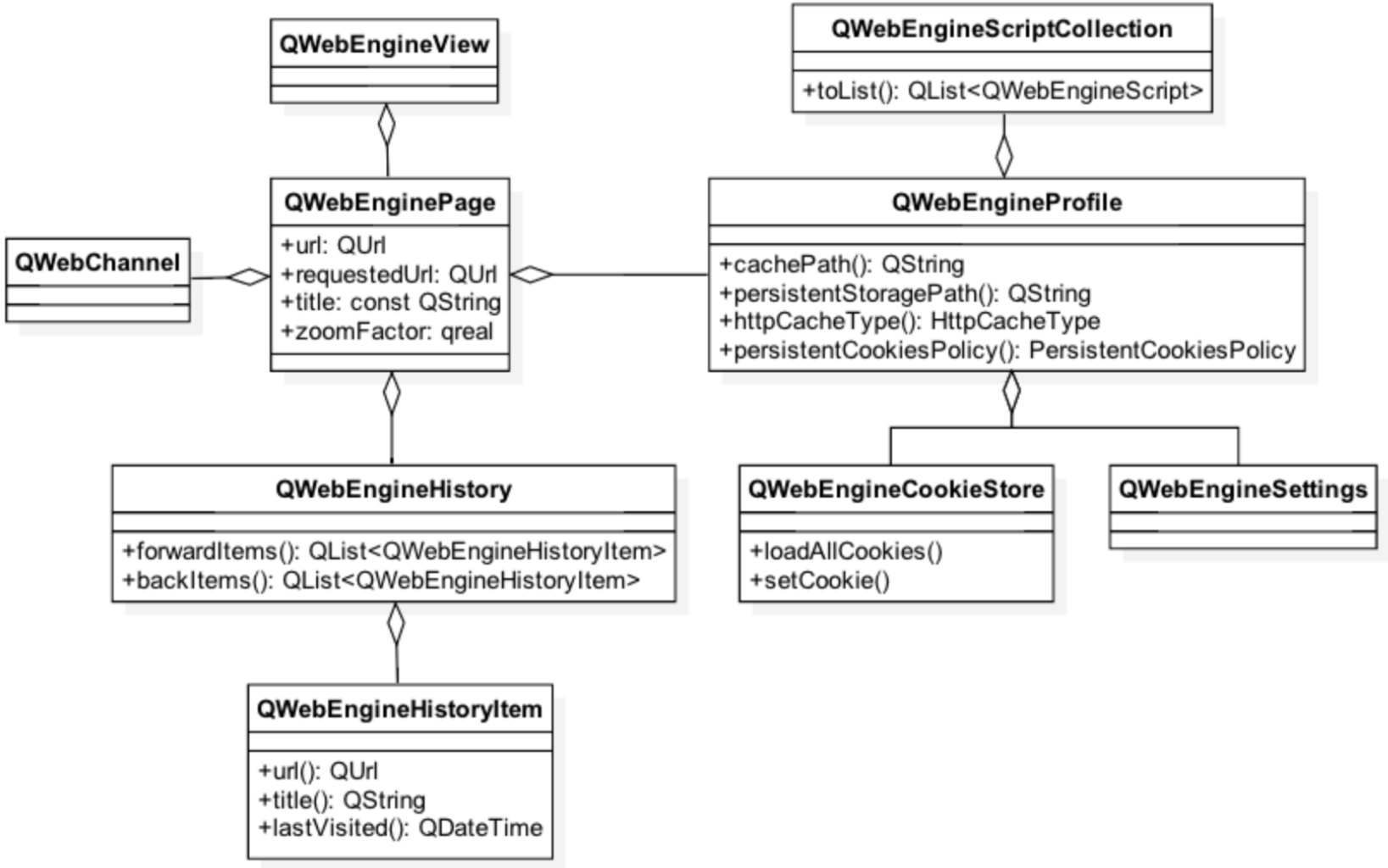
Qt WebEngine

- Provides a browser engine and web content interactions both in C++ and QML
  - Based on Google's Chromium project
- Qt also has Qt WebView module, which provides browser functionality in QML without a full web browser stack
  - Used in mobile platforms
- Chromium project
  - has cross-platform focus
  - provides browser on all major desktop platforms and Android
- Many features in Chromium work out-of-the-box without requiring Qt code
  - Platform/OS adaptation
  - Multimedia
  - HTML5 features such as WebRTC

Qt WebEngine

- Essential features in addition to web rendering
  - JS execution
  - Page conversion to HTML or to the plain text
  - Storage and cache management
  - Navigation history
  - Exposing Qt objects to JavaScript engine
- GUI process separated from the WebEngine process, taking care of page rendering and JS execution
  - Functions, resulting in inter-process communication, are asynchronous by nature
- Note that WebEngine Widgets use Qt Quick scene graph for composing page elements
  - Widgets rendering requires the scene graph rendering backend

Qt W
Widg



QWebEngineView and QWebEnginePage

- `QWebEngineView`
  - Allows editing and viewing web content
  - Functions: `load()`, `setUrl()`, `setHtml()`, `setContent()`, `setPage()`
  - Window management: `createWindow()` – called, when a new window requested in JavaScript

```
QWebEngineView *view = new QWebEngineView(parentObject);
view->load(QUrl("http://www.qt.io/"));
view->show();
```

- `QWebEnginePage`
  - Run JavaScript: `runJavaScript()`
  - Manage permissions: `setFeaturePermission()`
  - Trigger action: cut, paste, reload and bypass cache, redo, undo
  - Authentication: `authenticationRequired()`
  - Each page belongs to a profile with shared settings
  - Profile may be page-dedicated to allow private browsing

Other Essential Classes

- `QWebEngineSettings`
  - Font settings
  - Web attributes: auto load images, JS enabled, local storage enabled, JS can open windows
- `QWebEngineHistory`
  - Stores the navigation history in history items
  - Items may be accessed using currentItem(), backItems(), forwardItems()
- `QWebEngineProfile`
  - Profile shared by multiple pages
  - Access to settings
  - Storage path, cache path management
  - Cache types: memory or disk
- `QWebChannel`
  - Used to expose `QObjects` to HTML clients

Handling Asynchronous Functions

- Because of multi-process architecture, some of the web engine functions are asynchronous
- Asynchronous functions take a functor or lambda argument
- For example, `QWebEnginePage` allows to convert the web page to HTML or plaint text

```
void MainWindow::on_pushButton_clicked()
{
    QTextBrowser *textBrowser(ui->textBrowser);
    m_view->page()->toPlainText([textBrowser](const QString &result) {
        textBrowser->setText(result);
    });
}
```

Exposing Qt Object to JavaScript Engine

- `QWebChannel` allows exposing `QObject` properties, public slots and methods to HTML
  - Also property updates and signal emissions on the C++ side automatically transmitted to HTML clients

- Web channel requires a transport object for the communication between a C++ app and (possibly remote) HTML client
  - The transport object must implement an interface `QWebChannelAbstractTransport::sendMessage(const QJsonObject &msg)`
  - The implementation serializes the message and sends it to the client

Web Socket as Web Channel

- `QWebSocket` can be used as a transport channel
  - Implement `sendMessage()` using `QWebSocket::sendTextMessage();`
  - Emit the transport object `messageReceived()` signal in the slot, connected to `QWebSocket::textMessageReceived()` signal

- Web channel must be connected to the transport object
  - Connect a signal with the transport object argument to `QWebChannel::connectTo()` slot

Exposing Qt Object to JavaScript Engine

```
// Derives QWebChannelAbstractTransport
void WebSocketTransport::sendMessage(const QJsonObject &message)
{
    QJsonDocument doc(message);
    m_socket->sendTextMessage(QString::fromUtf8(doc.toJson(QJsonDocument::Compact)));
}

// A slot, connected to QWebSocket::textMessageReceived(const QString &msg) signal
void WebSocketTransport::textMessageReceived(const QString &messageData)
{
    QJsonParseError error;
    QJsonDocument message = QJsonDocument::fromJson(messageData.toUtf8(), &error);
    if (error.error) {
        qWarning() << "Parse error:" << messageData << error.errorString();
        return;
    }
    else if (!message.isObject()) {
        qWarning() << "Received JSON message that is not an object: " << messageData;
        return;
    }
    emit messageReceived(message.object(), this);
}
```

Exposing Qt Object to JavaScript Engine

- `QWebChannel` provides an API to register one or more `QObjects`
  - `channel.registerObject(QStringLiteral("myObject"), &object);`

- In the client side,
  - Create a web socket and provide callback functions `onError()`, `onClose()`, `onOpen()`
  - In `onOpen()`, create a web channel with a web socket and callback arguments
    - `new QWebChannel(socket, function(channel) { } )`
  - Registered objects are available through `channel.objects`
  - Essential functionality is provided by **qwebchannel.js**
  - The transport object is accessible through `navigator.qtWebChannelTransport`

Exposing Qt Object to JavaScript Engine

```
<script type="text/javascript" src="./qwebchannel.js"></script>

window.onload = function() {
    var socket = new WebSocket("ws://127.0.0.1:4321");

    socket.onclose = function() {
        console.error("Channel closed");
    };

    socket.onopen = function() {
        new QWebChannel(socket, function(channel) {

        // Access a property
        var propertyValue = channel.objects.myObject.propertyX;
        // Access a method
        channel.objects.myObject.somePublicMethod(propertyValue);
        // Access a signal
        channel.objects.myObject.someSignal.connect( function() { } );
    };
};
```

ex-exposing-objects

Questions and Answers

- How `QWebEngineView` is different from other widgets in terms of rendering?
- How do you access sub-frames?
- What is `QWenEngineProfile`?
- How cookies can be managed in Qt WebEngine?
- How can you use browser functionality in Qt applications in mobile platforms?
- How to expose Qt object to JS engine?

Summary

- Qt WebEngine allows having web browser functionality in applications
  - Both C++ widgets and QML types can be used
- In addition to page browsing, pages can be edited and converted to HTML or plain text
- Pages allow execution of JavaScript methods
- Qt objects may be exposed to JavaScript engine
  - Qt object features exposed in meta-object becomes accessible in the script engine

Thank You!

www.qt.io