KDE Development

Frameworks 5

For C++/Qt developers

KDE Frameworks Cookbook

The KDE Developers

July 11, 2022

Contents

1 Frameworks 5 **1**

1.1 History 2

2 Concurrent programming using the ThreadWeaver framework **5**

2.1 HelW olorld! 5

2.2 Adding ThreadWeaver to a project - an introduction to the Frameworks 5 build system 7

2.3 Hello World! with queueing multiple jobs 8

2.4 Doing things in a Sequence 11

2.5 Working title: Everything in moderation (and decorated) 16

3 KArchive **17**

3.1 Show me the code 17

3.2 Advanced usecases 18

3.2.1 Sending compressed data over networks 18

4 KItemModels **19**

4.1 Abstract 19

4.2 KBreadcrumbSelectionModel 19

4.3 KCheckableProxyModel 19

4.4 KDescendantsProxyModel 19

4.5 KLinkItemSelectionModel 20

4.6 KModelIndexProxyMapper 20

4.7 KRecursiveFilterProxyModel 20

4.8 KSelectionProxyModel 20

5 Spellchecking made easy **21**

5.1 Spellchecking in your QTextEdit 21

5.2 Language Detection in Sonnet 22

5.3 GUI Widgets provided by Sonnet 22

6 Reaching a wider audience **25**

6.1 Writing Messages 25

6.1.1 General Messages 25

6.1.2 Specialized Messages 27

6.1.3 Placeholder Substitution 29

7 Creating a new application **31**

7.1 Starting a new application from a template 31

7.2 Walking through the skeleton 34

7.2.1 main.cpp 34

7.2.2 BrightFuture 35

7.3 Plotting the future 36

7.4 Configuring the color 38

7.4.1 Enabling KConfig 38

7.4.2 Adding the capability to plot in different colors 39

7.4.3 Writing the configuration 40

7.4.4 Reading the configuration 41

8 Detect and Handle System Idling **43**

8.1 Using It 43

Chapter 1

Frameworks 5

KDE Frameworks 5 are a set of cross platform solutions that extend the functionality Qt offers. They are designed as drop-in Qt Addon libraries, enrich Qt as a development environment with functions that simplify, accelerate and reduce the cost of Qt development. Frameworks eliminate the need to reinvent key functionalities.

All frameworks come with quality promises, are developed in an open and welcoming environment, and are licensed under the Lesser Gnu Public License. By having each framework tailored to a specific use case, a framework can bring you the feature you need with a minimum of additional libraries.

Frameworks 5 consists of functional components and are structured in 'tiers' and 'categories'. The tiers give a structure for link-time dependencies. Tier 1 Frameworks can be used independently, while Tier 3 Frameworks can depend on other Tier 3 Frameworks and tiers below them. The catagories give information about the run-time dependencies, and are divided into the following three catagories:

- **Functional** frameworks have no runtime dependencies. For example, KArchive handles compression and decompression for many archive formats transparently and can be used as a drop-in library.
- **Integration** designates code that requires runtime dependencies for integration depending on what the OS or platform offers. For example, Solid supplies information on available hardware features and may require runtime components to deliver some of the data on some platforms.
- **Solutions** have mandatory runtime dependencies. For example, KIO (KDE Input/Output) offers a network-transparent virtual filesystem that lets users browse and edit files as if they were local, no matter where they are physically stored. And KIO requires kioslave daemons to function.

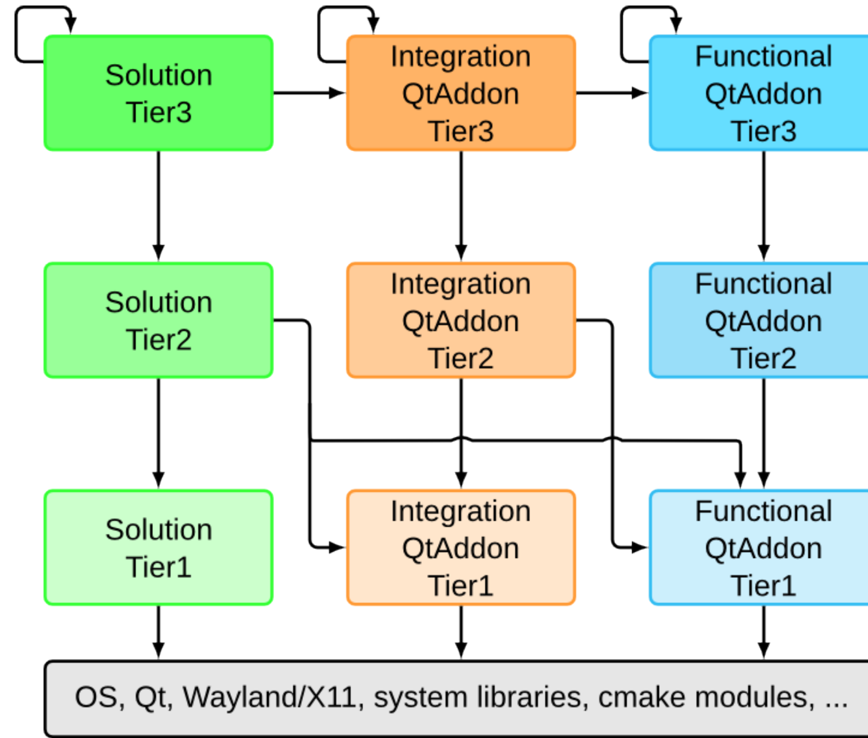

Figure 1.1: Frameworks 5 separates the KDE libraries into modules with clear dependencies.

The Frameworks are also separated by respecting core/gui distinctions and the different GUI technologies. So it is not uncommon to find a core, a gui and a widget module relating to a given Framework (e.g KConfigCore vs KConfigGui). This way third parties can use only the parts they need and avoid pulling unwanted dependencies on QtGui.

1.1 History

For over 15 years, the KDE libraries formed the common code base for (almost) all KDE applications. They provided a high-level functionality such as toolbars and menus, spell checking and file access. In that time 'kdelibs' was released and distributed as a single set of interconnected libraries. Through the KDE Frameworks efforts, these libraries have been methodically reworked into a set of independent, cross platform classes that now are available to all Qt developers.

The journey started at the Randa Meetings back in 2011, where porting KDE Platform 4 to Qt 5 was initiated. But as part of this effort, modularizing of

libraries, integrating portions properly into Qt 5 and modularizing was begun. Three years later, Frameworks 5 was released. Today you can save yourself the time and effort of repeating work that others have done, relying on over 50 Frameworks with mature, well tested code.

Chapter 2

Concurrent programming using the ThreadWeaver framework

2.1 HelW olorld!

Concurrent programming means creating applications that perform multiple operations at the same time. A common problem is that the user sees the application pause. A typical requirement is that an operation which may take an arbitrary amount of time because it is, for example, performing disk I/O, is scheduled for execution but immediately taken off the main thread of the application (the one that starts `main()`). To illustrate how this problem would be solved and to jump right into using ThreadWeaver, let's simulate this problem by printing *Hello World!* as the asynchronous payload.

```
#include <QtCore>
#include <ThreadWeaver/ThreadWeaver>

int main(int argc, char** argv)
{
    QCoreApplication app(argc, argv);

    using namespace ThreadWeaver;
    stream() << make_job( []() { qDebug() << "Hello World!"; } );
}
```

This short but complete program written in C++11 outputs the common greeting

to the command line.[1] It does so, however, from a worker thread managed by the global ThreadWeaver queue. The header file `ThreadWeaver/ThreadWeaver.h` included in line 2 contains the essential declarations needed to use the most common ThreadWeaver operations. The components used in this example are the global queue, a job and a queueing mechanism. The global queue is a singleton instance of the ThreadWeaver thread pool that is instantiated when it is first accessed after the application starts. A job represents "something" that should be executed asynchronously. In this case, the thing to execute is a C++ lambda function that prints the welcome message. The queueing mechanism used here is a queue stream, an API inspired by the *iostream* family of classes. ThreadWeaver builds on top of Qt, and similar to most Qt applications requires a QCoreApplication (or one of it's descendents) to exist throughout the lifetime of the application. Up to line 7, the program looks like any other Qt application.

To have the job lambda function called by one of the worker threads, a job is created that wraps it using the `make_job()` function. It is then handed to the queue stream. The queue stream will submit the jobs for execution when the queuing command is completed that is at the closing semicolon. Once the job is queued, one of the worker threads will automatically pick it up from the queue and execute it. `ThreadWeaver::Job` is the unit of execution handled by ThreadWeaver queues. Jobs are simple runnable types that perform one task, defined in their `run()` method. Some jobs wrap a lambda function as in this example or decorate other jobs. However implementing custom, reusable job classes is only a matter of writing a class that inherits `ThreadWeaver::Job` and re-implement its run method. The job that was created by `make_job()` in this example wraps the specified lambda function, and executes it when it is itself executed by a worker thread.

The program does not specify where the job should be executed, and not even when exactly. In a scenario where there would be many jobs waiting in the queue, execution of the new job would not be immediate. Which worker thread will be assigned the job is also undefined. The programmer gives up a bit of control over the details of execution, and in turns benefits from the automatic distribution of jobs amongst the available processors by the worker threads in the queue. Every program that links the ThreadWeaver library has access to a global queue for the execution of jobs. If no queue is specified when enqueueing a job, the global one will be used by default. Workers threads are allocated when needed by the queue. If the global pool is never accessed by an application, it will never be instantiated.

An application performing tasks in background threads should never exit while any of these operations is still in progress. In the case of ThreadWeaver, this means all jobs in the queue need to be either completed or dequeued and all worker threads idle before the application may exit. The global pool is in fact a QObject child of the `QCoreApplication` object instantiated in line 7. It will

[1]The examples are part of the ThreadWeaver source code and can be found at http://api.kde.org/frameworks-api/frameworks5-apidocs/threadweaver/html/index.html.

be deleted by the destructor of QCoreApplication. When it is destroyed, it will wait until all queued up jobs have completed. The program will thus wait in line 8 until the job has finished printing "Hello World!", and will then exit. The job was enqueued as a shared pointer, so memory management is taken care of. While this example was very much simplified, the described functionality already has many practical applications. For example, the many operations real-life applications need to perform at startup, like loading translations, icon resources et cetera, can be removed from the criticial path this way. In this case the operations usually need to be performed in a certain order and then handed over to the main thread. Solutions for that will be discussed in a later chapter.

2.2 Adding ThreadWeaver to a project - an introduction to the Frameworks 5 build system

Two standard questions occur to programmers when learning a new technology or toolkit as a programmer - how do I use it, and how do I add this module to and deploy it with my project. The answer to the second question requires at some knowledge about the build system used, and will be covered in this chapter. While it will use ThreadWeaver to explain the details, the workflow presented is generic and could be similarly applied when adding other KDE frameworks.

KDE frameworks use the CMake build system.[2] In essence, CMake is a generator of native project build instructions (Makefiles, for example) based on a project build description, the `CMakeLists.txt` file. CMake is common especially for C++ projects, and is used to build all of KDE software. The basic concepts are powerful, expressive and relatively easy to use. In addition, CMake is portable and generates build instructions for all relevant target platforms including not just Linux, but also OSX and Windows. This portability supports the goal of KDE and its frameworks to be available from a single source on as many platforms as possible. In the following steps, the essential bits of the complete CMakeLists.txt file for ThreadWeaver's HelloWorld example are going to be explained. The real world relevance of this use case is to build an application that uses and links a KDE framework, in this case ThreadWeaver.

```
5 cmake_minimum_required(VERSION 2.8.12)
6 find_package(ECM 1.1.0 REQUIRED NO_MODULE)
```

The first two lines define a minimum CMake version and make sure the extra CMake modules (ECM) used by the KDE project are detected by CMake. These two lines are not required, but it is a good idea to have them. Specifying a minimum CMake version at the beginning of the file prevents cryptic, hard to understand errors that may be caused by an older installed CMake version trying

[2] http://www.cmake.org

to parse the file any further. Similarly, ECM would be automatically detected if it is installed, but by explicitly looking for it, a clear error message is triggered if it cannot be found. However these two lines are just in preparation for the next bits that are more specific to the projects.

```
12 find_package(KF5ThreadWeaver ${KF5_VERSION} REQUIRED)
```

The `find_package` statement detects the ThreadWeaver include files and libraries and provides them so that they can later be used to build and link concrete targets, like libraries or applications. Because the `find_package` statement marks the framework as required, the statement will fail if ThreadWeaver cannot be detected by CMake. In this case, make sure the framework is properly installed, including the development package that usually contains the header files. On failure to detect ThreadWeaver, CMake will abort and not generate any makefiles.

```
17 # Define the project name
18 project(HelloWorld)
19 # Add the HelloWorld executable and link the ThreadWeaver
20 # library to it
21 add_executable(ThreadWeaver_HelloWorld HelloWorld.cpp)
22 target_link_libraries(ThreadWeaver_HelloWorld KF5::ThreadWeaver)
```

The last snippet defines the actual meat of the project. It specifies the project name to be `HelloWorld`, and adds an executable named `ThreadWeaver_HelloWorld` that is built from one source file, `HelloWorld.cpp`. The last line uses the `target_link_libraries` command to specify that to build the `ThreadWeaver_HelloWorld` executable, it should link the ThreadWeaver libraries. The libraries are specified using a scoped named variable, `KF5::ThreadWeaver`. This variable has been defined by the earlier `find_package` command. Every KDE framework defines a named variable like that that should be used to link the respective libraries.

2.3 Hello World! with queueing multiple jobs

The first example showed nothing that would have required multiple threads to print *Hello World!*, and also did not mention anything about at what time jobs get deleted. Object life span is of course a crucial questions when programming in C++. So of what type is the value that is returned by `make_job` in the first example?

The returned object is of type `JobPointer`, which is a `QSharedPointer` to `Job`. When `make_job` is executed, it allocates a `Job` that will later execute the C++ lambda function, and then embeds it into a shared pointer. Shared pointers

count references to the object pointer they represent, and delete the object when the last reference to it is destroyed. In a way, they are single-object garbage collectors. In the example, the new job is immediately handed over to the queue stream, and no reference to it is kept by `main()`. This approach is often called "fire-and-forget jobs". The queue will process the job and forget about it when it has been completed. It will then definitely get deleted automatically, even though the programmer does not necessarily know exactly when. It could happen (and in the case of ThreadWeaver jobs commonly does) deeply in the bowels of Qt event handling when the last event holding a reference to the job gets destroyed. The gist of it is that from the programmers point of view, it is not necessary to keep a reference to a job and delete it later. With that in mind, no further memory management is required in the HelloWorld example, and the program is complete.

Fire-and-forget jobs are not always the right tool. For example, if a job is retrieving and parsing some data, the application needs to access the data once the job is complete. For that, the programmer could implement a custom job class.

```
33 class QDebugJob : public Job {
34 public:
35     QDebugJob(const char* message = 0) : m_message(message) {}
36 protected:
37     void run(JobPointer, Thread*) {
38         qDebug() << m_message;
39     }
40 private:
41     const char* m_message;
42 };
```

The `QDebugJob` class simply prints a message to `qDebug()` when it is executed. To implement such a custom job class, it is inherited from `ThreadWeaver::Job`. By overloading the `run()` method, the "payload", the operation performed by the job, is being defined. The parameters to the run method are the job as the queue sees it, and the thread that is executing the job. The first parameter may be surprising. The reason that there may be a difference between the job that the queue sees and `this` is that jobs may be decorated, that means wrapped in something else that waddles and quacks like a job, before being queued. How this works will be explained later, what is important to keep in mind for now is not to assume to always find `this` in the queue.

```
46 int main(int argc, char** argv)
47 {
48     QCoreApplication app(argc, argv);
49     // Allocate jobs as local variables:
50     QDebugJob j1("Hello");
```

```
51      QDebugJob j2("World!");
52      JobPointer j3(new QDebugJob("This is..."));
53      Job* j4 = new QDebugJob("ThreadWeaver!");
54      // Queue the Job using the default Queue stream:
55      stream() << j1 << j2 // local variables
56           << j3 // a shared pointer
57           << j4; // a raw pointer
58      // Wait for finish(), because job is destroyed
59      // before the global queue:
60      Queue::instance()->finish();
61  }
```

This time, in the `main()` function, four jobs in total will be allocated. Two of them as local variables (j1 and j2), one (j3) dynamically and saved in a `JobPointer`, and finally j4 is allocated on the heap with `new`. All of them are then queued up for execution in one single command. Wait, what? Right. Local variables, job pointers and raw pointers are queued the same way and may be mixed and matched using the stream operators. When a local variable is queued, a special shared pointer will be used to hold it which does not delete the object when the reference count reaches zero. A `JobPointer` is simply a shared pointer. A raw pointer will be considered a new object and automatically wrapped in a shared pointer and deleted when it goes out of scope. Even though three different kinds of objects are handed over to the stream, in all three cases the programmer does not need to put special consideration into memory management and the object life cycles.

Now before executing the program, pause for a minute and think about what you expect it to print.

```
World!
This is...
Hello
ThreadWeaver!
```

Four jobs are being queued all at the same time, that is when the `stream()` statement closes. Assuming there is more than one worker thread, the order of execution of the jobs is undefined. The strings will be printed in arbitrary order. In case this comes as a surprise, it is important to keep in mind that by default, there is no relation between jobs that defines their execution order. This behaviour is in line with how thread pools normally work. In ThreadWeaver, there are ways to influence the order of execution by declaring dependencies between them or aggregating multiple jobs into collections or sequences. More on that later.

Before the end of `main()`, the application will block and wait for the queue to finish all jobs. This was not needed in the first HelloWorld example, so why is it

necessary here? As explained there, the global queue will be destroyed when the `QCoreApplication` object is destroyed. If `main()` would exit before j1 and j2 have been executed, it's local variables including j1 and j2 would be destroyed. In the destructor of `QCoreApplication` the queue would wait to finish all jobs, and try to execute j1 and j2, which have already been destructed. Mayhem would ensue. When using local variables as jobs, make sure that they have been completed before destroying them. The `finish()` method of the queue guarantees that it no more holds references to any jobs that have been executed.

2.4 Doing things in a Sequence

The time when an application starts, especially one that needs to load quite some data, is usually one of contention. Translations need to be loaded and resources like icons and images initialized. As the application matures, more and more of such tasks are piled on to it. It will have to check for updates from a server, and load a greeting of the day to the user. Eventually, the application will take ages to load, users will tweet about how they are making coffee while it comes up, and the programmers will start to find a solution.

The application will come up a lot faster if it defers as many tasks as possible while it creates and shows the user interface, and also takes as many as possible of the startup tasks of an application off the main thread. The main thread is the one that runs when `main()` is entered, and in which the user interface lives. Everything that slows down or intermittendly blocks the main thread may be experienced by the user as the user interface being sluggish or hung. This is a common use case where concurrent programming can help.

But ... this is also one of the examples where standard thread pools fail. The startup tasks commonly need to be done in a certain order and are of different priority, and also should not be all tackled by the application process at the same time. For example, the applications icons and translations may be needed first and urgently, where the information on available updates can still be processed a couple of seconds later. There are ways around this that are rather cumbersome, like using timers to queue up some tasks later or using chains of functions that queue up new tasks when one group is done. The following example will illustrate some aspects of how ThreadWeaver comes with the necessary tools to specify the order of tasks, on application startup and otherwise. The following `main()`[3] function allocates a main widget and an object of type `ViewController` that takes care of the startup tasks.

```
7  int main(int argc, char *argv[])
8  {
9      QApplication a(argc, argv);
```

[3] See `examples/HelloInternet` in the ThreadWeaver repository.

```
10      MainWidget w;
11      ViewController v(&w);
12      w.show();
13      a.exec();
14  }
```

The example application shows an image that it eventually loads from the network, and a caption for it. In the constructor of `ViewController`, the startup operations need to be kicked off. The operations in this example are

- to load a placeholder image that is shown while the application loads the image and caption from the network,
- to load the post that contains the caption, but only the URL of the image to show,
- and then, once the image URL is known, to finally load the image from the network and display it.

The application's user interface will be shown right away, even before step 1 has been completed. Let's assume that the three steps need to be done in order, not in parallel.

The important aspect is to do as little as possible in the constructor, considering that it is called from the main thread. Creating jobs and queueing them is not expensive however, so the constructor focuses on that and then returns.

```
21  {
22      connect(this, SIGNAL(setImage(QImage)),
23          mainwidget, SLOT(setImage(QImage)));
24      connect(this, SIGNAL(setCaption(QString)),
25          mainwidget, SLOT(setCaption(QString)));
26      connect(this, SIGNAL(setStatus(QString)),
27          mainwidget, SLOT(setStatus(QString)));
28
29      using namespace ThreadWeaver;
30      auto s = new Sequence;
31      *s << make_job( [=]() { loadPlaceholderFromResource(); } )
32         << make_job( [=]() { loadPostFromTumblr(); } )
33         << make_job( [=]() { loadImageFromTumblr(); } );
34      stream() << s;
35  }
```

Remember the assumption that the three startup steps have to be performed in order. The new thing here is that instead of queueing individual jobs, the constructor creates a `Sequence`, and then adds jobs to that. A sequence has the jobs performed by the thread pool in the order they have been added to it.

The jobs each simply call a member function of `ViewController` when being executed. ThreadWeaver's execution logic guarantees that the next job is only executed after the previous one has been finished. Because of that, only one of the member functions will be called at a time, and they will be called in the order the jobs have been added to the sequence.

Since only one of the member functions will be called at a time, there is no need for further synchronization of access to the member variables of `ViewController`. This raises the question of how the controller submits new captions, statuses and images to the main widget. It would be a mistake to simply call member functions of the main widget from the methods of `ViewController`, since these are executed from a worker thread. The controller submits update by using Qt signals that are connected to corresponding slots in the main widget. The parameters of the signals are passed by value, not by reference or pointers, making use of the implicit sharing built into Qt to avoid copying. This approach relies on the fact that the reference counting of Qt's implicit-sharing mechanism is thread safe.

```
44 void ViewController::loadPlaceholderFromResource()
45 {
46     QThread::msleep(500);
47     showResourceImage("IMG_20140813_004131.png");
48     emit setStatus(tr("Downloading post..."));
49 }
```

The method `loadPlaceholderFromResource()` implements the first step, to load an image from a resource that acts as a place holder until the real images has been downloaded. It cheats to appear busy by first sleeping for a short while. While it does so, the user interface will already appear to the user, with a blank background. It then emits a signal to make the main widget show a status message that indicates the program is downloading the post.

The method is called from the worker thread that executes the job, not the main thread. When the signal is emitted, Qt notices that sender and receiver are not in the same thread at the time, and sends the signal asynchroneously. The receiver will not be called from the thread executing `loadPlaceholderFromResource()`, instead it will be invoked from the event loop of the main thread. That means there is no shared data between the controller and the main widget for processing the signal, and no further serialization of access to the QString variable holding the status text is necessary.

Once the method returns and the job executing it completes, the next job of the sequence will be unlocked. This causes the method `loadPostFromTumblr()` to be executed by a worker thread. This method illustrates the convenience built into Qt to process data present in Open Standard formats (XML, in this case), even though this won't be discussed here in detail.[4] If processing the data turns

[4]The example uses the Tumblr API version 1.

out to be expensive, the user interface will not be blocked by it, since it is not performed by the main thread.

```
53 void ViewController::loadPostFromTumblr()
54 {
55     const QUrl url(m_apiPostUrl);
56
57     auto const data = download(url);
58     emit setStatus(tr("Post downloaded..."));
59
60     QDomDocument doc;
61     if (!doc.setContent(data)) {
62         error(tr("Post format not recognized!"));
63     }
64
65     auto textOfFirst = [&doc](const char* name) {
66         auto const s = QString::fromLatin1(name);
67         auto elements = doc.elementsByTagName(s);
68         if (elements.isEmpty()) return QString();
69         return elements.at(0).toElement().text();
70     };
71
72     auto const caption = textOfFirst("photo-caption");
73     if (caption.isEmpty()) {
74         error(tr("Post does not contain a caption!"));
75     }
76     emit setCaption(caption);
77     auto const imageUrl = textOfFirst("photo-url");
78     if (imageUrl.isEmpty()) {
79         error(tr("Post does not contain an image!"));
80     }
81
82     m_fullPostUrl = attributeTextFor(doc, "post", "url-with-slug");
83     if (m_fullPostUrl.isEmpty()) {
84         error(tr("Response does not contain URL with slug!"));
85     }
86     m_imageUrl = QUrl(imageUrl);
87     showResourceImage("IMG_20140813_004131-colors-cubed.png");
88     emit setStatus(tr("Downloading image..."));
89     QThread::msleep(500);
90 }
```

In case an error occurs, the method invokes another method called `error()`. `error()` indicates the problem to the user by setting a status messages in the main widget. But it also apparently aborts the execution of the sequence, as the code assumes it does not continue after calling it.

Figure 2.1: Hello Internet

```
126 void ViewController::error(const QString &message)
127 {
128     showResourceImage("IMG_20140813_004131-colors-cubed.png");
129     emit setCaption(tr("Error"));
130     emit setStatus(tr("%1").arg(message));
131     throw ThreadWeaver::JobFailed(message);
132 }
```

`error()` shows a different placeholder image, and emits the status message to the main widget. It then raises an exception of type `ThreadWeaver::JobFailed`, which will be caught by the worker thread executing the current job. The worker thread sets the status of the job to a failed state. Specific to sequences (because only sequences know the order of the execution of their elements), this will cause the sequence to abort the execution of it's remaining elements. Raising the exception will abort the processing of the job, but not terminate the worker thread. Leaking any other type of exception than `ThreadWeaver::Exception` from the `run()` method of a job is considered a runtime error. The exception will not be caught by the worker thread, and the application will terminate.

The example illustrates the steps necessary to perform concurrent operations in a certain order. It also shows how a specialized object (`ViewController`, in this case) can handle the data shared between the sequential operations, how to submit data and status information back to the user interface, and how to signal error conditions from job execution.

2.5 Working title: Everything in moderation (and decorated)

Let's put the features that have been described so far and a few more that as of yet haven't been mentioned, and create a comprehensive example program. The example program calculates thumbnails for images. It will take a number of image files and, in separate steps implemented as individual jobs, load them from disk, convert them from raw data to QImages, scale the images to thumbnails, and finally save them to disk. This problem may not be most imaginative use of concurrent programming techniques, but it does demonstrate a number of practical problems. For example, the operations for each single image contain elements that are file system I/O bound and elements that are CPU bound. For large numbers of images, it has to deal with a trade-off of memory usage and CPU utlilization. Less obvious, there are also expectations on the order of execution of the jobs, so that the interface provides the user with visible feedback of the progress of the operations. An application of this kind also should provide features of load management and reduce it's own generated system load if the system is "stressed" by other processes. A web server implementation or a video coding program will have to solve similar issues to provide optimal throughput without overloading the system.

Chapter 3

KArchive

When you are storing large amounts of data, how do you archive it in a easy way from within your code? The KArchive framework provides a quick and easy way to do this from within Qt apps.

While Qt5 provides the QZipWriter and QZipReader classes, these are limited only to Zips. KArchive on the other hand supports a wide array of formats such as p7zip, tar and ar archives, giving you the flexibility of choosing the formats which fit your project.

3.1 Show me the code

Here's a simple 'Hello World' example of KArchive.

```
46      // Create a zip archive
47      KZip archive(QStringLiteral("hello.zip"));
48
49      // Open our archive for writing
50      if (archive.open(QIODevice::WriteOnly)) {
51
52          // The archive is open, we can now write data
53          archive.writeFile(QStringLiteral("world"),                   // File name
54                            QByteArray("The whole world inside a hello."), // Data
55                            0100644,                                    // Permissions
56                            QStringLiteral("owner"),                    // Owner
57                            QStringLiteral("users"));                   // Group
58
59          // Don't forget to close!
60          archive.close();
61      }
```

More files can be added by subsequent calls to writeFile(). You also add folders to your zip by using the writeDir call as follows :

```
archive.writeDir(QStringLiteral("world dir"));
```

Full API docs can be found here

3.2 Advanced usecases

3.2.1 Sending compressed data over networks

KArchive also supports reading and writing compressed data to devices such as buffers or sockets via the KCompressionDevice class allowing developers to save bandwidth while transmitting data over networks.

A quick example of the KCompressionDevice class can be summed up as:

```
70      // Open the input archive
71      KCompressionDevice input(&file, false, KCompressionDevice::BZip2);
72      input.open(QIODevice::ReadOnly);
73
74      QString outputFile = (info.completeBaseName() + QStringLiteral(".gz"));
75
76      // Open the new output file
77      KCompressionDevice output(outputFile, KCompressionDevice::GZip);
78      output.open(QIODevice::WriteOnly);
79
80      while(!input.atEnd()) {
81          // Read and uncompress the data
82          QByteArray data = input.read(512);
83
84          // Write data like you would to any other QIODevice
85          output.write(data);
86      }
87
88      input.close();
89      output.close();
```

Chapter 4

KItemModels

4.1 Abstract

KItemModels is a set of classes built for or on top of Qt's model view system. It contains a collection of additional proxy models and other utilities to help make complex tasks around models simpler. The following chapter will go through all of them one by one

4.2 KBreadcrumbSelectionModel

The `KBreadcrumbSelectionModel` is a selection model that ensures that some or all parents of items in trees are selected when a given item is selected. `KBreadcrumbSelectionModel` makes creating breadcrumb navigation bar easy with this.

4.3 KCheckableProxyModel

The `KCheckableProxyModel` adds checkable capability to an `QAbstractItemModel` without having to modify the model itself and implement the right parts of data, setData and flags methods. The checkable proxy model also works nicely together with the `KSelectionProxyModel` to show the items checked off.

4.4 KDescendantsProxyModel

`KDescendantsProxyModel` flattens a tree model into a list with the possibility to still make it visually appear like a tree by indentation or by showing the parent's

4.5 KLinkItemSelectionModel

`KLinkItemSelectionModel` makes it possible to share a selection between multiple views that has different proxy models in between the root model and the view

4.6 KModelIndexProxyMapper

`KModelIndexProxyMapper` facilitates mapping between two different branches of proxy models on top of the same base root model.

4.7 KRecursiveFilterProxyModel

Filtering a tree model where the child items are of interest, `QSortFilterProxyModel` is not the right thing. `QSortFilterProxyModel` does not look at children if a parent is filtered out. `KRecursiveFilterProxyModel` goes through the tree and includes a item and all its parents.

4.8 KSelectionProxyModel

`KSelectionProxyModel` Convenience filtering model to just show the items that are included by a `QItemSelectionModel`

Chapter 5

Spellchecking made easy

Sonnet is a useful framework provided by KDE for software developers who want to solve the problem of spellchecking in text editors. It has a plugin based architechture with support for HSpell, Enchant, ASpell and HUNSPELL plugins. It even supports automated language detection, based on a combination of different algorithms.

5.1 Spellchecking in your QTextEdit

Sonnet can be easily integrated into your QTextEdit as follows:

```
59      QTextEdit *textEdit = new QTextEdit;
60      textEdit->setText("This is a sample buffer. Whih this thingg will "
61                        "be checkin for misstakes. Whih, Enviroment, govermant. Whih."
62                       );
63
64      Sonnet::SpellCheckDecorator *installer = new Sonnet::SpellCheckDecorator(textEdit);
65      installer->highlighter()->setCurrentLanguage("en");
```

Sonnet::SpellCheckDecorator can also be extended in various ways to spell check text that is formatted differently, for example in emails.

```
34  class MailSpellCheckDecorator : public Sonnet::SpellCheckDecorator
35  {
36  public:
37      MailSpellCheckDecorator(QTextEdit *edit)
38          : Sonnet::SpellCheckDecorator(edit)
39      {}
```

```
40
41  protected:
42      bool isSpellCheckingEnabledForBlock(const QString &blockText) const Q_DECL_OVERRIDE
43      {
44          qDebug() << blockText;
45          return !blockText.startsWith(QLatin1Char('>'));
46      }
47  };
```

So, you can use MailSpellCheckDecorator in exactly the same way as you would use SpellCheckDecorator, but with the added functionality that MailSpellCheckDecorator will ignore quoted parts of a email.

5.2 Language Detection in Sonnet

Sonnet can determine the difference between ~75 languages for a given string. It is based off a perl script origionaly written by Maciej Ceglowski called Languid. His script used a two-part heuristic to determine language. First the text is checked for the scripts it contains, next for each set of languages using those scripts a n-gram frequency model of a given language is compared to a model of the text. The most similar language model is assumed to be the language. If no language is found an empty string is returned.

Here you see a simple example of language detection using the GuessLanguage class from Sonnet:

```
GuessLanguage languageGuesser;
QString lang = languageGuesser.identify("My awesome text");
```

5.3 GUI Widgets provided by Sonnet

Sonnet also provides some GUI widgets that can be used by Qt applications to configure settings in Sonnet; for example Qt applications can use the DictionaryComboBox class from Sonnet to get a QComboBox that can configure the dictionary used by Sonnet.

```
37  void TestDialog::check(const QString &buffer)
38  {
39      Sonnet::Dialog *dlg = new Sonnet::Dialog(
40          new BackgroundChecker(this), 0);
41      connect(dlg, SIGNAL(done(QString)),
42              SLOT(doneChecking(QString)));
```

```
43      dlg->setBuffer(buffer);
44      dlg->show();
45  }
```

The ConfigDialog class from Sonnet provides a more advanced configuration dialog to configure settings such as whitelisting words, skipping run-together words as well as enabling or disabling auto detection of the language.

Chapter 6

Reaching a wider audience

A excellent way of reaching a wider audience with your software is by localizing it. The KDE community provides the ki18n framework to do this by leveraging gettext underneath. While Qt provides tr, ki18n is much much more powerful than tr, and offers writing 3 broad categories of writing messages: General Messages, Specialized Messages, Placeholder Substitution, while also providing functionality to include user interface markers to provide better context to translators.

6.1 Writing Messages

Most messages can be internationalized with simple `i18n*` calls, which are described in the “General Messages” section. A few messages may require treatment with `ki18n*` calls, and when this is needed is described in the “Special Messages” section. Argument substitution in messages is performed using the familiar Qt syntax `%<number>`, but there may be some differences.

6.1.1 General Messages

General messages are wrapped with `i18n*` calls. These calls are *immediate*, which means that they return the final localized text (including substituted arguments) as a `QString` object, that can be passed to UI widgets.

The most frequent message type, a simple text without any arguments, is handled like this:

```
QString msg = i18n("Just plain info.");
```

The message text may contain arbitrary Unicode characters, and the source file *must* be UTF-8 encoded. Ki18n supports no other character encoding.

If there are some arguments to be substituted into the message, `%<number>` placeholders are put into the text at desired positions, and arguments are listed after the string:

```
QString msg = i18n("%1 has scored %2", playerName, score);
```

Arguments can be of any type for which there exists an overloaded `KLocalizedString::subs` method. Up to 9 arguments can be inserted in this fashion, due to the fact that `i18n` calls are realized as overloaded templates. If more than 9 arguments are needed, which is extremely rare, a `ki18n*` call must be used.

Sometimes a short message in English is ambiguous to translators, possibly leading to a wrong translations. Ambiguity can be resolved by providing a context string along the text, using the `i18nc` call. In it, the first argument is the context, which only the translator will see, and the second argument is the text which the user will see:

```
QString msg = i18nc("player name - score", "%1 - %2", playerName, score);
```

In messages stating how many of some kind of objects there are, where the number of objects is inserted at run time, it is necessary to differentiate between *plural forms* of the text. In English there are only two forms, one for number 1 (singular) and another form for any other number (plural). In other languages this might be more complicated (more than two forms), or it might be simpler (same form for all numbers). This is handled properly by using the `i18np` plural call:

```
QString msg = i18np("%1 image in album %2", "%1 images in album %2",
                    numImages, albumName);
```

The plural form is decided by the first integer-valued argument, which is `numImages` in this example. In rare cases when there are two or more integer arguments, they should be ordered carefully. It is also allowed to omit the plural-deciding placeholder, for example:

```
QString msg = i18np("One image in album %2", "%1 images in album %2",
                    numImages, albumName);
```

or even:

```
QString msg = i18np("One image in album %2", "More images in album %2",
                    numImages, albumName);
```

If the code context is such that the number is always greater than 1, the plural call must be used nevertheless. This is because in some languages there are different plural forms for different classes of numbers; in particular, the singular form may be used for numbers other than 1 (e.g. those ending in 1).

If a message needs both context and plural forms, this is provided by `i18ncp` call:

```
QString msg = i18ncp("file on a person", "1 file", "%1 files", numFiles);
```

In the basic `i18n` call (no context, no plural) it is not allowed to put a literal string as the first argument for substitution. In debug mode this will even trigger a static assertion, resulting in compilation error. This serves to prevent misnamed calls: context or plural frequently needs to be added at a later point to a basic call, and at that moment the programmer may forget to update the call name from `i18n` to `i18nc/p`.

Furthermore, an empty string should never be wrapped with a basic `i18n` call (no `i18n("")`), because in translation catalog the message with empty text has a special meaning, and is not intended for client use. The behavior of `i18n("")` is undefined, and there will be some warnings in debug mode.

6.1.2 Specialized Messages

There are some situations where `i18n*` calls are not sufficient, or are not convenient enough. One obvious case is if more than 9 arguments need to be substituted. Another case is if it would be easier to substitute arguments later on, after the line with the i18n call. For cases such as these, `ki18n*` calls can be used. These calls are *deferred*, which means that they do not return the final translated text as `QString`, but instead return a `KLocalizedString` instance which needs further treatment. Arguments are then substituted one by one using `KLocalizedString::subs` methods, and after all arguments have been substituted, the translation is finalized by one of `KLocalizedString::toString` methods (which return `QString`). For example:

```
KLocalizedString ks;
case (reportSource) {
    SRC_ENG: ks = ki18n("Engineering reports: %1"); break;
    SRC_HEL: ks = ki18n("Helm reports: %1"); break;
    SRC_SON: ks = ki18n("Sonar reports: %1"); break;
    default: ks = ki18n("General report: %1");
}
QString msg = ks.subs(reportText).toString();
```

`subs` methods do not update the `KLocalizedString` instance on which they are invoked, but return a copy of it with one argument slot filled. This allows to use `KLocalizedString` instances as a templates for constructing final texts, by supplying different arguments.

Another use for deferred calls is when special formatting of arguments is needed, like requesting the field width or number of decimals. `subs` methods can take these formatting parameters. In particular, arguments should not be formatted in a custom way, because `subs` methods will also take care of proper localization (e.g. use either dot or comma as decimal separator in numbers, etc):

```
// BAD (number not localized):
QString msg = i18n("Rounds: %1", myNumberFormat(n, 8));
// Good:
QString msg = ki18n("Rounds: %1").subs(n, 8).toString();
```

Like with `i18n`, there are context, plural, and context-plural variants of `ki18n`:

```
ki18nc("No function", "None").toString();
ki18np("File found", "%1 files found").subs(n).toString();
ki18ncp("Personal file", "One file", "%1 files").subs(n).toString();
```

`toString` methods can be used to override the global locale. To override only the language of the locale, `toString` can take a list of languages for which to look up translations (ordered by decreasing priority):

```
QStringList myLanguages;
...
QString msg = ki18n("Welcome").toString(myLanguages);
```

This section describes how to specify the translation *domain*, a canonical name for the catalog file from which `*i18n*` calls will draw translations. But `toString` can always be used to override the domain for a given call, by supplying a specific domain:

```
QString trName = ki18n("Georgia").toString("country-names");
```

Relevant here is the set of `ki18nd*` calls (`ki18nd`, `ki18ndc`, `ki18ndp`, `ki18ndcp`), which can be used for the same purpose, but which are not intended to be used directly. Please refer to this page to check when these calls should be made.

6.1.2.1 Dynamic Contexts

Translators are provided with the capability to script translations, such that the text changes based on arguments supplied at run time. For the most part, this feature is transparent to the programmer. However, sometimes the programmer may help in this by providing a *dynamic* context to the message, through `KLocalizedString::inContext` methods. Unlike the static context, the dynamic context changes at run time; translators have the means to fetch it and use it to script the translation properly. An example:

```
KLocalizedString ks = ki18nc("%1 is user name; may have "
                             "dynamic context gender=[male,female]",
                             "%1 went offline");
if (knownUsers.contains(user) && !knownUsers[user].gender.isEmpty()) {
    ks = ks.inContext("gender", knownUsers[user].gender);
}
QString msg = ks.subs(user).toString();
```

Any number of dynamic contexts, with different keys, can be added like this. Normally every message with a dynamic context should also have a static context, like in the previous example, informing the translator of the available dynamic context keys and possible values. Like `subs` methods, `inContext` does not modify the parent instance, but returns a copy of it.

6.1.3 Placeholder Substitution

Hopefully, most of the time `%<number>` placeholders are substituted in the way one would intuitively expect them to be. Nevertheless, some details about substitution are as follows.

Placeholders are substituted in one pass, so there is no need to worry about what will happen if one of the substituted arguments contains a placeholder, and another argument is substituted after it.

All same-numbered placeholders are substituted with the same argument.

Placeholders directly index arguments: they should be numbered from 1 upwards, without gaps in the sequence, until each argument is indexed. Otherwise, error marks will be inserted into message at run time (when the code is compiled in debug mode), and any invalid placeholder will be left unsubstituted. The exception is the plural-deciding argument in plural calls, where it is allowed to drop its placeholder, in either the singular or the plural text.

If none of the arguments supplied to a plural call is integer-valued, an error mark will be inserted into the message at run time (when compiled in debug mode).

Integer arguments will be by default formatted as if they denote an amount, according to locale rules (thousands separation, etc.) But sometimes an integer is a numerical identifier (e.g. port number), and then it should be manually converted into `QString` beforehand to avoid treatment as amount:

```
i18n("Listening on port %1.", QString::number(port));
```

6.1.3.1 User Interface Markers

In the same way there exists a HIG (Human Interface Guidelines) document for the programmers to follow, translators should establish HIG-like convention for their language concerning the forms of UI text. Therefore, for a proper translation, the translator will need too know not only what does the message mean, but also where it figures in the UI. E.g. is the message a button label, a menu title, a tooltip, etc.

To this end a convention has been developed among KDE translators, which programmers can use to succinctly describe UI usage of messages. In this convention, the context string starts with an *UI marker* of the form `@<major>:<minor>`, and may be followed by any other usual context information, separated with a single space:

```
i18nc("@action:inmenu create new file", "New");
```

The major and minor component of the UI marker are not arbitrary, but are drawn from a table which can be found here.

For much more detail, see http://api.kde.org/frameworks-api/frameworks5-apidocs/ki18n/html/prg_guide.html

Chapter 7

Creating a new application

You have an awesome idea. The idea which will change the world, which will bring everybody a bright future. This idea needs to be implemented *now*, so you sit down and do it. Your toolkit of choice is Qt, what else?

There are many ways to start a new Qt application. One of them is using the tool `kapptemplate`, which generates a fresh skeleton of an application you can then fill with all the goodness your idea brings.

7.1 Starting a new application from a template

So you run `kapptemplate` and start the wizard. First you have to choose which template to use. We use the "Minimal C++ KDE Frameworks" one. This will get us started and open up a bunch of nice opportunities coming from KDE Frameworks. More about that later.

We enter the name of our new application "BrightFuture" and continue the wizard.

Now we just need to enter some basic data about the application, the initial version number, author, and where the code should be stored. This will usually already be neatly pre-filled.

Now continue and finish the wizard and you have the initial code ready for your new application.

Before you compile the code, we highly recommend to first create a build folder that will be separated from your source folder. That's because when you start compiling the application, the build system will create lots of files during the compilation and the folder with your source code could quickly become overpopulated with files. This way you'll have a clean separation between source code and the compiled binary files.

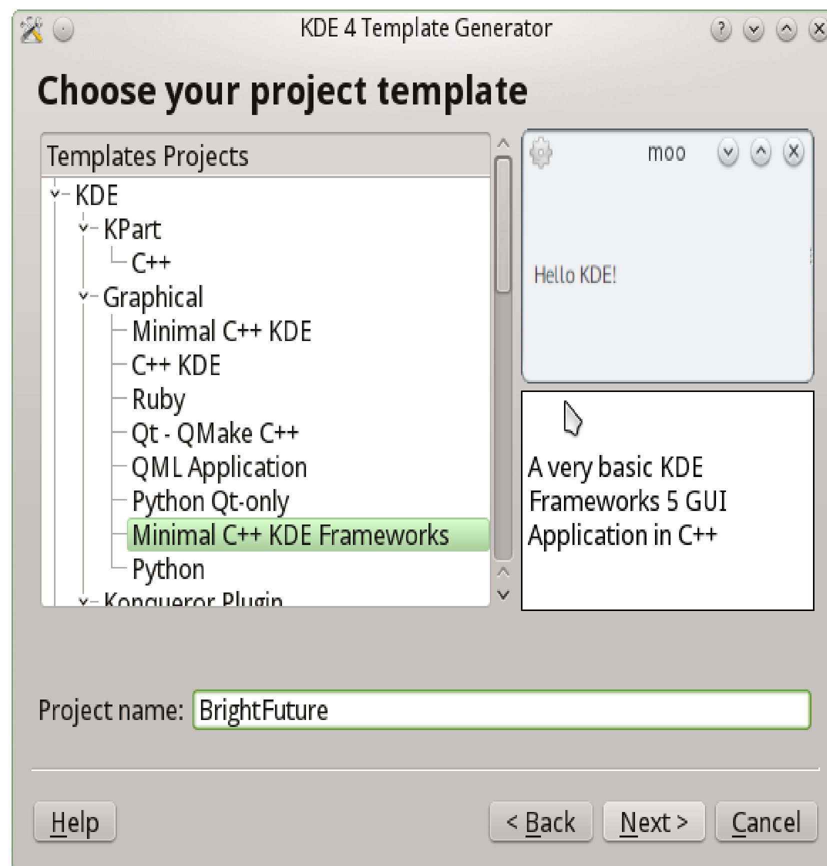

Figure 7.1: Choose application template

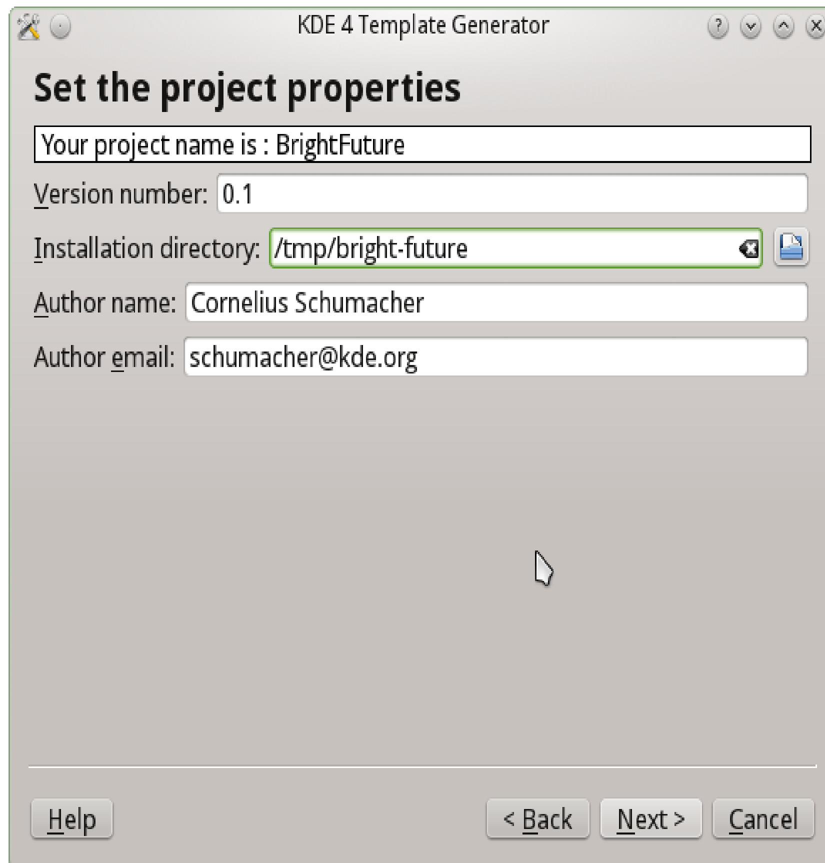

Figure 7.2: Enter data

Go to the code folder, create a “build” folder and cd into it

```
mkdir build
cd build
```

Now compile it with

```
cmake ..
make
```

Run it with

```
src/brightfuture
```

and there you are. Greetings from KDE to your new application.

7.2 Walking through the skeleton

Let’s have a look at what was generated there and walk through the initial code.

7.2.1 main.cpp

The starting point is main.cpp. That’s where the application is set up. The first line of the main function creates an application object:

```
31     QApplication application(argc, argv);
```

This is straightforward, but there is one important thing to notice, especially if you have not seen KDE applications before. We use a `QApplication`; that’s with a `Q` not a `K`. So no special setup is needed anymore for writing applications with KDE Frameworks. It’s just a Qt application, and you can later add whatever you need whenever you want.

The scope of your idea of course doesn’t stop at language barriers, so the template conveniently sets up internationalization of the texts in your application under a dedicated translation domain:

```
35     KLocalizedString::setApplicationDomain("brightfuture");
```

The next step is to set up some basic information about the application, so that this can be shown to users and wherever else this is useful:

```
38      KAboutData aboutData( QStringLiteral("brightfuture"),
39                            i18n("Simple App"),
40                            QStringLiteral("0.1"),
41                            i18n("A Simple Application written with KDE Frameworks"),
42                            KAboutLicense::GPL,
43                            i18n("(c) 20013-2014, Cornelius Schumacher <schumacher@kde.org>"));
44
45      aboutData.addAuthor(i18n("Cornelius Schumacher"),i18n("Author"), QStringLiteral("schumacher@k
46      aboutData.setProgramIconName("brightfuture");
```

This makes use of the data you entered in the wizard before. Note that it uses the `i18n` function to translate all strings visible to users. This comes from the KDE Framework `k18n`.

The `KAboutData` class comes from the KDE Framework `kcoreaddons`.

As the next step, the command line is parsed, so users can get help about the use of the program from the command line, information about author and version and whatever options BrightFuture will need:

```
49      QCommandLineParser parser;
50      parser.addHelpOption();
51      parser.addVersionOption();
52      aboutData.setupCommandLine(&parser);
53      parser.process(application);
54      aboutData.processCommandLine(&parser);
```

Finally we show the application's main window and give control to the user:

```
58      BrightFuture *appwindow = new BrightFuture;
59      appwindow->show();
60      return application.exec();
```

7.2.2 BrightFuture

The main window is implemented in the class `BrightFuture`. Let's have a look.

The header brightfuture.h is minimal:

```
31  /**
32   * This class serves as the main window for BrightFuture.  It handles the
33   * menus, toolbars and status bars.
34   *
35   * @short Main window class
36   * @author Your Name <mail@example.com>
```

```
37  * @version 0.1
38  */
39 class BrightFuture : public QMainWindow
40 {
41     Q_OBJECT
42 public:
43     /**
44      * Default Constructor
45      */
46     BrightFuture();
47
48     /**
49      * Default Destructor
50      */
51     virtual ~BrightFuture();
52
53 private:
54     // this is the name of the root widget inside our Ui file
55     // you can rename it in designer and then change it here
56     Ui::mainWidget m_ui;
57 };
```

It defines a window inherited from `QMainWindow` and adds a main widget `Ui::mainWidget m_ui;`, which is defined in the Qt Designer file brightfuture.ui.

The implementation brightfuture.cpp brings the application to life in its constructor:

```
27     QWidget *widget = new QWidget(this);
28     setCentralWidget(widget);
29     m_ui.setupUi(widget);
```

This is standard Qt code. We will add a little bit more here later.

7.3 Plotting the future

We know the future is bright, so let our application plot it. KDE Frameworks comes with the framework `KPlotting`, which is able to do simple plots. See the KPlotting API for more information. We will use it to plot a set of data points in our main window.

To make use of the framework, declare that you are using it in your main CMakeLists.txt file. Simply add `Plotting` to the `find_package` statement for the KDE Frameworks libraries (it uses `KF5` as a shortcut):

```
find_package(KF5 REQUIRED COMPONENTS
    CoreAddons
    I18n
    Plotting
)
```

You also have to link to the library in the CMakeLists.txt file in the `src` directory where the source files of the application are defined, and how they are linked to the required libraries. Add `KF5::Plotting` to the `target_link_libraries` statement there:

```
target_link_libraries( brightfuture
    Qt5::Widgets
    KF5::CoreAddons
    KF5::I18n
    KF5::Plotting
)
```

Now we can write the actual code to plot the future. We add that to the constructor of the main window and replace the code, which was generated by the template generator there:

```
30     KPlotWidget *plot = new KPlotWidget(this);
31     setCentralWidget(plot);
32 
33     plot->setLimits(-1, 11, -1, 40);
34 
35     KPlotObject *po =
36         new KPlotObject(Qt::white, KPlotObject::Bars, 2);
37     po->setBarBrush(QBrush(Qt::green, Qt::Dense4Pattern));
38 
39     float y = 1;
40     for (float x = 1; x <= 10; x += 1) {
41         po->addPoint(x, y);
42         y *= 1.5;
43     }
44 
45     plot->addPlotObject(po);
46 
47     plot->update();
```

That's all. Here is the plot of the future:

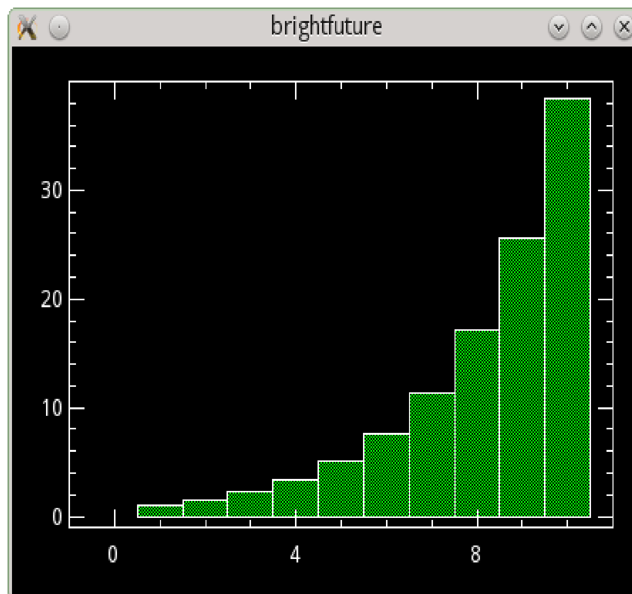

Figure 7.3: sample plot

7.4 Configuring the color

The future is bright, but everybody has a different preference for its color. So let's make the color of the future configurable.

KDE Frameworks offers `KConfig`, which is a framework for reading and writing configuration data. We will make use of it in our application to save the color of the plot we created in the previous section.

7.4.1 Enabling KConfig

As the first we need to add the framework to the main CMakeList.txt, so that includes and libraries become available:

```
18 find_package(KF5 REQUIRED COMPONENTS
19     CoreAddons
20     I18n
21     Plotting
22     Config
23 )
```

Then we need to link to the `ConfigGui` library in the CmakeList.txt file in the `src` directory to be able to access the functions `KConfig` provides:

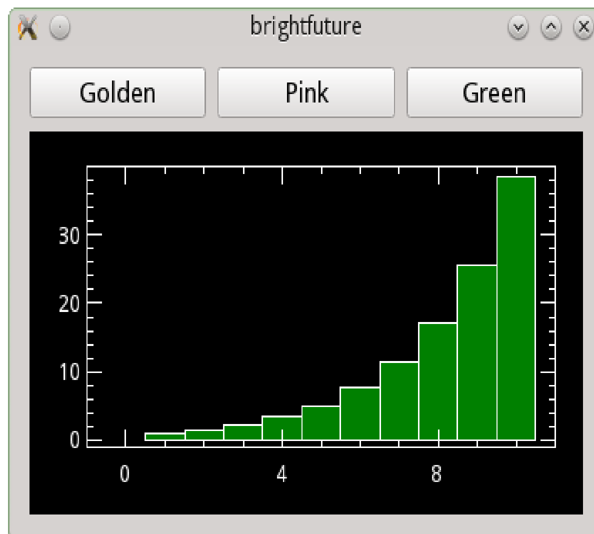

Figure 7.4: Three-color future

```
 9  target_link_libraries( brightfuture
10      Qt5::Widgets
11      KF5::CoreAddons
12      KF5::I18n
13      KF5::Plotting
14      KF5::ConfigGui
15  )
```

`KConfig` provides two libraries: `KConfigCore` and `KConfigGui`. The core library contains the basic functionality. The GUI library adds support for data type used in GUIs. We want to store a color, which is a GUI type, that is why we link to `KConfigGui`.

7.4.2 Adding the capability to plot in different colors

To be able to make the color configurable, `brightfuture` first needs to be able to plot in different colors. We simply do that by adding three buttons, which each call a separate slot setting the colors to green, golden, or pink.

This code is straight-forward Qt code. It is in brightfuture.h and brightfuture.cpp. Have a look there to see the details. We will focus on the configuration code now.

7.4.3 Writing the configuration

We need to classes for dealing with configuration data, `KSharedConfig` and `KConfigGroup`, so we add the include statements for them at the top of the brightfuture.cpp file:

```
27 #include <KSharedConfig>
28 #include <KConfigGroup>
```

`KSharedConfig` represents a configuration. It is the main class, which provides access to configuration groups and takes care of storing, reading, and writing configuration data.

`KConfigGroup` represents a named configuration group. This is the object you need to actually read and write configuration data. It takes a name, which is used to group the configuration in the configuration files.

Now that we have the classes available, we just need to make use of them:

```
91 void BrightFuture::plotGoldenFuture()
92 {
93     KConfigGroup config(KSharedConfig::openConfig(), "colors");
94     config.writeEntry("plot", QColor("gold"));
95     plotFuture();
96 }
```

This is the function which is called when pressing one of the color buttons. It sets the color and then calls the function doing the actual plot. The magic happens in the first two lines of the function.

The first line creates the `KConfigGroup` object, which is used to write the configuration. It uses the application-wide shared configuration object, which is retrieved by the `KSharedConfig::openConfig()` call. The second parameter is the name of the group, where the configuration should be stored.

The second line writes the configuration value we want to store. We simply call `writeEntry` on the group object, give it a name of our choice for the configuration option, and pass the color as the object to store. `KConfig` does the magic to figure out how to deal with a `QColor` object in the configuration file behind the scenes.

By default configuration is stored in a INI-style text file in the directory `~/.config/brightfuturerc`:

```
[colors]
plot=255,215,0
```

The name of the configuration file is derived from the application name defined by `KAboutData` in the main.cpp file:

```
38      KAboutData aboutData( QStringLiteral("brightfuture"),
39              i18n("Simple App"),
40              QStringLiteral("0.1"),
41              i18n("A Simple Application written with KDE "
42              "Frameworks"),
43              KAboutLicense::GPL,
44              i18n("(c) 2013-2014, "
45              "Cornelius Schumacher <schumacher@kde.org>"));
46
47      aboutData.addAuthor(i18n("Cornelius Schumacher"),
48                      i18n("Author"),
49                      QStringLiteral("schumacher@kde.org"));
50      aboutData.setProgramIconName("brightfuture");
```

7.4.4 Reading the configuration

Now the final step is to read the configuration on startup of the application, so that the choice of the user is remembered.

This is done in the `plotFuture` function:

```
107 void BrightFuture::plotFuture()
108 {
109     KConfigGroup config(KSharedConfig::openConfig(), "colors");
110     QColor color = config.readEntry("plot", QColor("green"));
111     m_plot_object->setBarBrush(QBrush(color, Qt::SolidPattern));
112     m_plot->update();
113 }
```

We get the “color” group from the configuration object for the application again and then call `readEntry` to read the value we wrote before. The second parameter `QColor("green")` is the default value which is used when no value can be found in the configuration file.

We can now start the application, click the “golden” button to change the color of the plot to gold, and the next time we start the application the plot is rendered golden at once.

That’s all we need. We have made the color of the future configurable and made it golden.

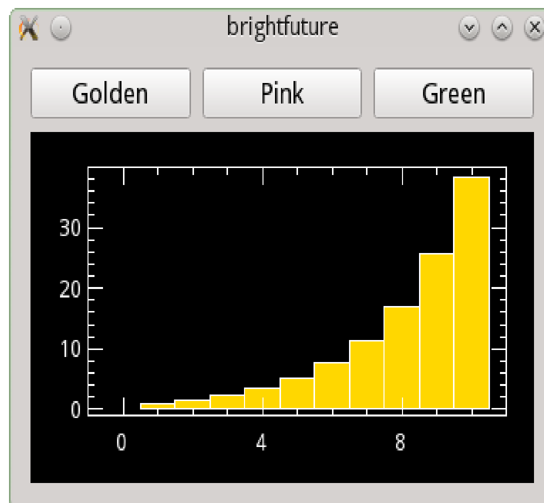

Figure 7.5: Golden future

Chapter 8

Detect and Handle System Idling

KIdleTime is a helper framework to get reporting information on idle time of the system. It is useful not only for finding out about the current idle time of the system, but also for getting notified upon idle time events, such as custom timeouts or user activity. It features:

- current idling time
- timeout notifications, to be emitted if the system idled for a specified time
- activity notifications, if the user resumes acting after an idling periode

8.1 Using It

For understanding how to use KIdleTime, we create a small testing application, called KIdleTest. This application initially waits for the first user action and afterwards registers some timeout intervals, and acts whenever the system idles for such a time. The KIdleTime framework provides a singleton KIdleTime, which provides us with all necessary signals and information about the idling status of the system. For our example, we start with connecting to the signals for user resuming from idling and for reaching timeouts that we will set ourselves:

```
26  KIdleTest::KIdleTest()
27  {
28      // connect to idle events
29      connect(KIdleTime::instance(), SIGNAL(resumingFromIdle()),
30              this, SLOT(resumeEvent()));
31      connect(KIdleTime::instance(), SIGNAL(timeoutReached(int,int)),
```

```
32              this, SLOT(timeoutReached(int,int)));
33
34      // register to get informed for the very next user event
35      KIdleTime::instance()->catchNextResumeEvent();
36
37      printf("Your idle time is %d\n", KIdleTime::instance()->idleTime());
38      printf("Welcome!! Move your mouse or do something to start...\n");
39  }
```

We also tell KIdleTime to notify us the very next time when the user acts. Note that this is actually only for the next time. If we were interested in further events, we had to invoke `catchNextResumeEvent()` again. Next, in our event listener for the user resume event, we add register a couple of idle intervals:

```
47  void KIdleTest::resumeEvent()
48  {
49      KIdleTime::instance()->removeAllIdleTimeouts();
50
51      printf("Great! Now stay idle for 5 seconds to get a nice message. From 10"
52             "seconds on, you can move your mouse to get back here.\n");
53      printf("If you will stay idle for too long, I will simulate your activity"
54             "after 25 seconds, and make everything start back\n");
55
56      KIdleTime::instance()->addIdleTimeout(5000);
57      KIdleTime::instance()->addIdleTimeout(10000);
58      KIdleTime::instance()->addIdleTimeout(25000);
59  }
```

If any of these idle intervals is reached, our initially registered `timeoutReached(...)` slot is invoked and we print out an appropriate message.

```
63  void KIdleTest::timeoutReached(int id, int timeout)
64  {
65      Q_UNUSED(id)
66
67      if (timeout == 5000) {
68          printf("5 seconds passed, stay still some more...\n");
69      } else if (timeout == 10000) {
70          KIdleTime::instance()->catchNextResumeEvent();
71          printf("Cool. You can move your mouse to start over\n");
72      } else if (timeout == 25000) {
73          printf("Uff, you're annoying me. Let's start again. I'm simulating your"
74                 "activity now\n");
75          KIdleTime::instance()->simulateUserActivity();
76      } else {
```

```
77            qDebug() << "OUCH";
78        }
79    }
```

From there on, depending on the reached idle interval, we go back to one of the former steps.

KDE Framewsorks 5

This book is mainly for C++/Qt developers, who want to extend the Qt capabilities, using KDE Frameworks.

Regular users of the software do not need this book. Those interested in programming might find it interesting to understand how the complex and richly featured software we use is created.

Discover a variety of frameworks and their usecases. And because nothing can replace real code, the book will guide you with several examples how to quickly obtain results.

Learn more about KDE at www.kde.org