KDE Development

Frameworks 5

For C++/Qt developers

KDE Frameworks Cookbook

The KDE Developers

Contents

1 Introduction to KF5 **1**

1.1 What are Frameworks? 1

1.2 History 2

2 KArchive **3**

2.1 Show me the code 3

2.2 Sending compressed data over networks 4

3 Kauth **5**

3.1 Introduction to Authorization 5

3.2 What is KAuth 5

3.3 Concepts 5

3.4 Phases breakdown 6

3.5 Creating Actions with KAuth 8

3.6 The Domain field 9

3.7 CMake macros and file format 9

4 Introduction to KConfig **10**

4.1 Design Essentials 10

4.2 The KConfig Class 11

4.3 Special Configuration Objects 11

4.4 KSharedConfig 13

4.5 KConfigGroup 13

4.6 Reading Entries 13

4.7 Writing Entries 14

4.8 KDesktopFile: A Special Case 14

4.9 KConfig XT 14

5 Ki18n: Reaching a wider audience **15**
5.1 Writing Messages 15
5.2 General Messages 15
5.3 Specialized Messages 17
5.4 Placeholder Substitution 19

6 KIdleTime: Detect and Handle System Idling **20**
6.1 Using Kidletime 20

7 KItemModels **22**
7.1 KBreadcrumbSelectionModel 22
7.2 KCheckableProxyModel 22
7.3 KDescendantsProxyModel 22
7.4 KLinkItemSelectionModel 22
7.5 KModelIndexProxyMapper 23
7.6 KRecursiveFilterProxyModel 23
7.7 KSelectionProxyModel 23

8 Sonnet: Spellchecking made easy **24**
8.1 Spellchecking in your QTextEdit 24
8.2 Language Detection in Sonnet 25
8.3 GUI Widgets provided by Sonnet 25

9 Concurrent programming using the ThreadWeaver framework **26**
9.1 HelW olorld! 26
9.2 Adding ThreadWeaver to a project - an introduction to the Frameworks 5 build system 28

10 Creating a new application **30**
10.1 Starting a new application from a template 30
10.2 Walking through the skeleton 32
10.2.1 main.cpp 32
10.2.2 BrightFuture 33
10.3 Plotting the future 34
10.4 Configuring the color 35
10.4.1 Enabling KConfig 35

10.4.2 Adding the capability to plot in different colors 36
10.4.3 Writing the configuration 36
10.4.4 Reading the configuration 38

Chapter 1

Introduction to KF5

1.1 What are Frameworks?

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.

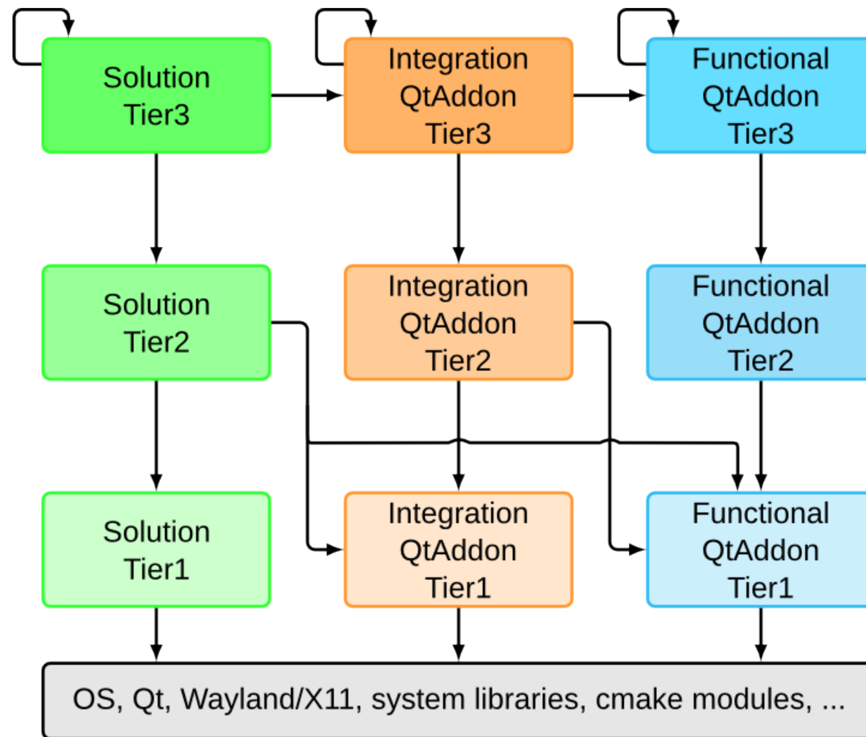
All frameworks come with quality promises, are developed in an open and welcoming environment, and are licensed under the Lesser Gnu Public License. By having each framework tailored to a specific use case, a framework can bring you the feature you need with a minimum of additional libraries.

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.

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.



1.2 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

KArchive

When you are storing large amounts of data, how do you archive it in an 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.

2.1 Show me the code

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

```
// Create a zip archive
    KZip archive(QStringLiteral("hello.zip"));

    // Open our archive for writing
    if (archive.open(QIODevice::WriteOnly)) {
        // The archive is open, we can now write data
        archive.writeFile(QStringLiteral("world"), // File name
                          QByteArray("The world inside a hello."), // Data
                          0100644, // Permissions
                          QStringLiteral("owner"), // Owner
                          QStringLiteral("users")); // Group

        // Don't forget to close!
        archive.close();
    }

    if (archive.open(QIODevice::ReadOnly)) {
        const KArchiveDirectory *dir = archive.directory();

        const KArchiveEntry *e = dir->entry("world");
        if (!e) {
            qDebug() << "File not found!";
            return -1;
        }
```

```
25          const KArchiveFile *f = static_cast<const KArchiveFile *>(e);
26          QByteArray arr(f->data());
27          qDebug() << arr; // the file contents
28
29          // To avoid reading everything into memory in one go,
30          // we can use createDevice() instead
31          QIODevice *dev = f->createDevice();
32          while (!dev->atEnd()) {
33              qDebug() << dev->readLine();
34          }
35          delete dev;
36      }
```

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.

2.2 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:

```
1  // Open the input archive
2      KCompressionDevice input(&file, false, KCompressionDevice::BZip2);
3      input.open(QIODevice::ReadOnly);
4
5      QString outputFile = (info.completeBaseName() + QLatin1String(".gz"));
6
7      // Open the new output file
8      KCompressionDevice output(outputFile, KCompressionDevice::GZip);
9      output.open(QIODevice::WriteOnly);
10
11     while (!input.atEnd()) {
12         // Read and uncompress the data
13         QByteArray data = input.read(512);
14
15         // Write data like you would to any other QIODevice
16         output.write(data);
17     }
18
19     input.close();
20     output.close();
```

Chapter 3

Kauth

3.1 Introduction to Authorization

When writing an application occasionally we want to access actions that require administrator access. This could include writing some configuration files that are owned by root, editing the system clock or other administrative tasks.

The obvious solution is to run the entire application as root, but this exposes a lot of potential security problems to the user. We want a way to run the main application as the normal user, yet still be able to authenticate and and run small parts as root or another user.

3.2 What is KAuth

KAuth is an authentication framework, that acts as a wrapper around lower-level libraries and tools. If you are planning to use KAuth, however, you won't have to care about what authentication system is the system you are targeting using: KAuth will take care of that on its own.

In addition, KAuth is also able to perform privilege elevation on restricted portions of code (helpers), giving the developer an efficient and easy to use pipe to communicate with them, and making them secure throughout authorization.

3.3 Concepts

There are a few concepts to understand when using KAuth. Much of those are carried from underlying APIs such as polkit, so if you are familiar with one of them you might as well skip this section.

- The 'authorization system' is an underlying framework (like polkit or Authorization Services), which KAuth interfaces with. KAuth's aim is to never make the developer

know or care about the underlying authorization system, however in this tutorial series this concept will come up quite often to explain better how KAuth works.

- An ''action" is a single task that needs authorization to be performed. Each action has an unique action identifier, which is a string in reverse domain name syntax, like ''org.kde.this.is.an.action". For example, if our example application needs to read a file the user has no privileges on, it would need an action like "org.kde.auth.xmpl.read". Please note that each action has to refer to a single task: this allows system administrators to fine tune the policies that allow users to perform your actions, and also a more secure way of locking down the privileged actions in your application.
- An ''action namespace" is the first part of the action identifier. In the command "org.kde.auth.xmpl.read", ''org.kde.auth.example" is the **action namespace**, ''read" is the **action name**. This is a very important concept when dealing with helpers and .actions files.
- ''Authorization" is a particular phase where the underlying authorization system performs the needed checks (and eventually asks the user its credentials in order to authorize him). Before any action is executed, the Authorization phase takes place. This is handled internally by KAuth: even if you are able to trigger this phase manually, most of the times you don't need to: KAuth will still execute an action only if the underlying authorization system allows its execution.
- ''Authentication"is an optional phase that takes place during authorization, if the policy for the action requests the user to input a credential to give him an explicit authorization. This phase is external and not handled by KAuth, but entirely by the underlying authorization system. It is, however, important for you to know something about it even if KAuth has no way to hijack the Authentication phase by design.
- ''Execution" happens **only** if the Authorization was successful: the execution might consist in a simple confirmation of the successful authorization, or eventually in the execution of a function in an helper.
- An ''Helper"is a separate application running as a privileged user (usually root), which is called upon execution if your action was attached to an helper. KAuth uses a completely transparent approach: IPC between your application and the helper itself is handled internally through an extremely simple API, and you won't even know that the helper is a separate application: spawning, killing and all the process handling is handled by KAuth.

3.4 Phases breakdown

Supposing that you want to use KAuth to perform a privileged operation and the action you are considering requires the user to authenticate (which is the most common use case of KAuth), the break down of phases would be:

- The user wants to perform some privileged task.

- The application creates an action for the task in question.
- The action is requested to be executed.
- The following steps are handed internally, either by KAuth or the underlying authorization system.
- *Authorization phase begins*
- The system detects that the user needs to authenticate to authorize the action.
- *Authentication phase begins*
- The user is requested to input his password, swipe his finger, press a button. . .
- *Authentication phase ends*
- If authentication was unsuccessful, the action is rejected.
- Otherwise, the system grants an explicit authorization to the user.
- *Authorization phase ends*
- If authorization was unsuccessful, the action is rejected.
- Otherwise, the action is executed.
- *Execution phase begins*
- A separate application is spawned by root, and the requested portion of code is executed.
- The helper code, immediately after starting, checks the authorization again to improve security, and also because some authorization systems delay the authorization phase in the beginning of the execution phase. If the helper is not authorized, the execution is aborted.
- If the caller is authorized, the helper executes the task.
- *Execution phase ends*
- The application receives the result of the execution from the helper.

This is how, concept-proof, KAuth works. However, please note that in your implementation you will have to deal with the pre-authorization phase only, since everything else is handled internally.

3.5 Creating Actions with KAuth

To increase the level of security, authorization systems require to register the actions together with the application installation, so that the authorized actions are all known to the system administrator. This means that if you're using KAuth you probably want to register some new actions in the system.

Many authorization systems are quite strict about action naming, hence staying compatible with all of them is a tricky task. To ensure maximum compatibility with all of them, in both action namespaces and action names, **use only lowercase letters and numbers**. Here comes a small example:

- "org.kde.auth.xmpl.read" **OK**
- "org.KDE.auth.xmpl.read" **NOT OK**
- "org.kde.auth.xmpl-1.read" **NOT OK**
- "org.kde.auth.xmpl.readFile" **NOT OK**
- "org.kde.auth.xmpl.readfile" **OK**

This is done by creating a ".actions" file, which is a standard INI files containing a set of new actions. This file is translatable, and if you're developing your project in KDE git, scripty will take care of updating it.

Each .actions file can contain an unlimited set of actions, provided that they **belong to the same action namespace**. This is extremely important.

The file has the following format:

[org.kde.auth.example.action]
Name=Example action
Description=The system is attempting to perform the example action
Policy=auth_admin
Persistence=session

The fields are defined as follows:

- **Title**: The action identifier
- **Name**: A human readable action name
- **Description**: This message will eventually be displayed to the user during the authentication phase, if any.
- **Policy**: The default policy for this action. It can be one of the following values: *"yes": the action should be allowed without requesting authentication* "no": the action should be always denied, without requesting authentication *"auth_self": the action will be authorized if the user will authenticate as himself* "auth_admin": the action will be authorized if the user will authenticate as a system administrator

- **Persistence**: this field is optional and takes effect only if the authorization system supports it and **Policy** is either "auth_admin" or "auth_self". It defines the persistence of the explicit authorization granted by the user through authentication. It can be one of the following values: *''session'': the authorization persists until the user logs out* "always": the authorization will persist indefinitely

3.6 The Domain field

.actions files can have a special group, **[Domain]**, under which you can give out some more information about the action namespace you're defining. This is how it looks (all fields are optional):

[Domain] Name=The KAuth example series
Icon=kauth-example
URL=http://techbase.kde.org/

The fields are defined as follows:

- **Name**: Usually the name defining the application which is going to use this namespace
- **Icon**: An icon name, that will be shared among all the actions
- **URL**: Home page of your organization

3.7 CMake macros and file format

Once you defined the actions in your file (remember you can define an unlimited number of actions in an .actions file, provided that they all belong to the same namespace, for example org.kde.auth.example.*), KAuth provides a CMake macro to register the actions in the system. From your CMakeLists.txt, supposing your file is named org.kde.auth.example.actions, you would do:

```
kauth_install_actions(org.kde.auth.example org.kde.auth.example.actions)
```

This macro has the following syntax:

```
kauth_install_helper_files(<namespace_id> <actions definition file>)
```

Where namespace_id is the namespace where you defined your actions, in this case org.kde.auth.example.

Chapter 4

Introduction to KConfig

This is based on the KConfig tutorial on Techbase

This tutorial looks at the KDE configuration data system, starting with an overview of the design fundamentals from an application developer's point of view. It then looks at the classes relevant to application development one by one.

4.1 Design Essentials

KConfig is designed to abstract away the concept of actual storage and retrieval of configuration settings behind an API that allows easy fetching and setting of information. Where and in what format the data is stored is not relevant to an application using KConfig. Using Kconfig keeps all KDE applications consistent in their handling of configurations while sparing application authors from the work of building such a system on their own. This eliminates many errors.

A KConfig object represents a single configuration object. Each configuration object is referenced by its unique name and may be actually read from multiple local or remote files or services. Each application has a default configuration object associated with it and there is also a global configuration object.

These configuration objects are broken down into a two level hierarchy: groups and keys. A configuration object can have any number of groups and each group can have one or more keys with associated values.

Values stored may be of any number of data types. They are stored and retrieved as the objects themselves. For example, a `QColor` object is passed to a config object directly when storing a color value and when retrieved a `QColor` object is returned. Applications themselves therefore generally do not have to perform serialization and deserialization of objects themselves.

4.2 The KConfig Class

The `KConfig` object is used to access a given configuration object. There are a number of ways to create a config object:

```
// An example of a KConfig object
// a plain old read/write config object
KConfig config("myapprc");

// a specific file in the filesystem
// currently must be an INI style file
KConfig fullPath("/etc/kderc");

// not merged with global values
KConfig globalFree( "localsrc", KConfig::NoGlobals );

// not merged with globals or the $KDEDIRS hierarchy
KConfig simpleConfig( "simplerc", KConfig::SimpleConfig );
```

The KConfig object created on line 3 is a regular config object. We can read values from it, write new entries and ask for various properties of the object. This object will be loaded from the config resource as determined by `QStandardPaths`, meaning that every instance of the `myapprc` object in each of the directories in the config resource hierarchy will be merged to create the values seen in this object. This is how system wide and per-user/group profiles are generated and supported and it all happens transparently to the application itself.

The hierarchy of directories searched for configuration is defined by `$XDG_CONFIG_DIRS`, which is defined in the XDG Base Directory Specification. Qt supports this specification in `QStandardPaths`.

On line 7 we open a specific local file, this case `/etc/kderc`. This performs no merging of values and expects an INI style file.

Line 10 sees the creation of a configuration object that is not merged with the user global configuration object, while the configuration file on line 13 is additionally not merged with any files in the `$XDG_CONFIG_DIRS` hierarchy. This can noticeably improve performance in the case where one is simply reading values out of a simple configuration for which global values are not meaningful.

4.3 Special Configuration Objects

Each application has its own configuration object. This object uses the name provided to `KAboutData` with "rc" appended as its name. So an app named `myapp` would have the default configuration object of `myapprc` (located in `$XDG_CONFIG_HOME`, which is `~/.config` by default). This configuration file can be retrieved in this way:

```
#include <KSharedConfig>

MyClass::MyClass()
{
    KSharedConfig::Ptr config = KSharedConfig::openConfig();
}
```

This actually uses `KSharedConfig`, which is a ref-counted shared `KConfig` object. More about that in a later section.

The default configuration object for the application is accessed when no name is specified when creating a `KConfig` object. So we could also do this instead, but it would be slower because it would have to parse the whole file again:

```
#include <KConfig>

MyClass::MyClass()
{
    KConfig config;
}
\end{lstlisting}

\section{Commonly Useful Methods}

To save the current state of the configuration object we call the
\texttt{sync()} method. This method is also called when the object is
destroyed. If no changes have been made or the resource reports itself
as non-writable (such as in the case of the user not having write
permissions to the file) then no disk activity occurs. \texttt{sync()}
merges changes performed concurrently by other processes. Local changes
have priority, though.

To ensure that we have the latest values from disk, call
\texttt{reparseConfiguration()} which calls \texttt{sync()} and then
reloads the data from disk.

To prevent the config object from saving already-made modifications to
disk, call \texttt{markAsClean()}. A particular use case for this is
rolling back the configuration to the on-disk state by calling
\texttt{markAsClean()} followed by \texttt{reparseConfiguration()}.

Listing all groups in a configuration object is as simple as calling
\texttt{groupList()} as in this code snippet:

\begin{lstlisting}
KSharedConfig::Ptr config = KSharedConfig::openConfig();

foreach ( const QString& group, config->groupList() ) {
    qDebug() << "next group:" << group;
}
```

4.4 KSharedConfig

The `KSharedConfig` class is a reference counted pointer to a `KConfig`. It thus provides a way to reference the same configuration object from multiple places in your application without the extra overhead of separate objects or concerns about synchronizing writes to disk even if the configuration object is updated from multiple code paths.

Accessing a `KSharedConfig` object is as easy as this:

```
KSharedConfig::Ptr config = KSharedConfig::openConfig("ksomefilerc");
```

`openConfig()` take the same parameters as `KConfig`'s constructors do, allowing one to define which configuration file to open, flags to control merging and non-`config` resources.

`KSharedConfig` is generally recommended over using `KConfig` itself.

4.5 KConfigGroup

Now that we have a configuration object, the next step is to actually use it. First define which group of key/value pairs we wish to access in the object. We do this by creating a KConfigGroup object:

```
KConfig config;
KConfigGroup generalGroup( &config, "General" );
KConfigGroup colorsGroup = config.group( "Colors" ); // or a bit differently
```

You can pass `KConfig` or `KSharedConfig` objects to `KConfigGroup`.

Config groups can be nested as well:

```
KConfigGroup subGroup1( &generalGroup, "LessGeneral" );
KConfigGroup subGroup2 = colorsGroup.group( "Dialogs" );
```

4.6 Reading Entries

With a `KConfigGroup` object in hand reading entries is now quite straight forward:

```
QString accountName = generalGroup.readEntry( "Account",
                                              QString() );
QColor color = colorsGroup.readEntry( "background",
                                      Qt::white );
QStringList list = generalGroup.readEntry( "List",
                                           QStringList() );
QString path = generalGroup.readPathEntry( "SaveTo",
                                           defaultPath );
```

In the example above, one can mix reads from different `KConfigGroup` objects created on the same `KConfig` object. The read methods take the key, which is case sensitive,

as the first argument and the default value as the second argument. This argument controls what kind of data, e.g. a color in line 74 above, is to be expected as well as the type of object returned. The returned object is wrapped in a `QVariant` to make this magic happen.

If no such key currently exists in the configuration object, the default value is returned instead. If there is a localized (e.g.translated into another language) entry for the key that matches the current locale, that is returned.

4.7 Writing Entries

Setting new values is similarly straightforward:

```
1 generalGroup.writeEntry( "Account", accountName );
2 generalGroup.writePathEntry( "SaveTo", savePath );
3 colorGroup.writeEntry( "background", color );
4 generalGroup.config()->sync();
```

Note the use of `writePathEntry` and how the type of object we use, such as `QColor` on line 86, dictates how the data is serialized. Additionally, once done writing entries, `sync()` must be called on the config object for it to be saved to disk. We can also simply wait for the object to be destroyed, which triggers an automatic `sync()` if necessary.

4.8 KDesktopFile: A Special Case

When is a configuration file not a configuration file? When it is a desktop file. These files, which are essentially configuration files, which are used to describe entries for application menus, mimetypes, plugins and various services.

When accessing a .desktop file, one should instead use the `KDesktopFile` class which, while a `KConfig` class offering all the capabilities described above, offers a set of methods designed to make accessing standard attributes of these files consistent and reliable.

4.9 KConfig XT

There is a way to make certain use cases of KConfig easier, faster and more reliable: KConfig XT. In particular, for main application or plugin configuration objects and when syncing configuration dialogs and other interfaces with these values, KConfig XT can help immensely. It also simultaneously documents the configuration options available, which makes every sysadmin and system integrator that uses KDE software that much more happy.

Read more about Using KConfig XT.

Chapter 5

Ki18n: 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.

5.1 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.

5.2 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 must be of a type for which an overloaded `KLocalizedString::subs` method exists. 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.

5.3 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 instead return a copy of it with one argument slot filled. This permits us to use `KLocalizedString` instances as 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.

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.

5.4 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));
```

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 the online version of this guide.

Chapter 6

KIdleTime: 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

6.1 Using Kidletime

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:

```
1  KIdleTest::KIdleTest()
2  {
3      // connect to idle events
4      connect(KIdleTime::instance(), &KIdleTime::resumingFromIdle,
5      this, &KIdleTest::resumeEvent);
6      connect(KIdleTime::instance(), qOverload<int, int>
7      (&KIdleTime::timeoutReached),
8      this, &KIdleTest::timeoutReached);
9
10     // register to get informed for the very next user event
11     KIdleTime::instance()->catchNextResumeEvent();
```

```
12      printf("Your idle time is %d\n", KIdleTime::instance()->idleTime());
13      printf("Welcome!! Move your mouse or do something to start...\n");
14  }
```

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:

```
1   void KIdleTest::resumeEvent()
2   {
3       KIdleTime::instance()->removeAllIdleTimeouts();
4
5       printf("Great! Now stay idle for 5 seconds to get a nice message.
6       From 10 seconds on, you can move your mouse to get back here.\n");
7       printf("If you will stay idle for too long, I will simulate your
8       activity after 25 seconds, and make everything start back\n");
9
10      KIdleTime::instance()->addIdleTimeout(5000);
11      KIdleTime::instance()->addIdleTimeout(10000);
12      KIdleTime::instance()->addIdleTimeout(25000);
13  }
```

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

```
1   oid KIdleTest::timeoutReached(int id, int timeout)
2   {
3       Q_UNUSED(id)
4
5       if (timeout == 5000) {
6           printf("5 seconds passed, stay still some more...\n");
7       } else if (timeout == 10000) {
8           KIdleTime::instance()->catchNextResumeEvent();
9           printf("Cool. You can move your mouse to start over\n");
10      } else if (timeout == 25000) {
11          printf("Uff, you're annoying me. Let's start again.
12          I'm simulating your activity now\n");
13          KIdleTime::instance()->simulateUserActivity();
14      } else {
15          qDebug() << "OUCH";
16      }
17  }
```

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

Chapter 7

KItemModels

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

7.1 KBreadcrumbSelectionModel

The `KBreadcrumbSelectionModel` is a selection model to ensure that the parents of items in trees are selected when a given item is selected. `KBreadcrumbSelectionModel` makes creating a breadcrumb navigation bar easy with this.

7.2 KCheckableProxyModel

The `KCheckableProxyModel` adds checkable capability to a `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.

7.3 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

7.4 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

7.5 KModelIndexProxyMapper

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

7.6 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.

7.7 KSelectionProxyModel

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

Chapter 8

Sonnet: 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.

8.1 Spellchecking in your QTextEdit

Sonnet can be easily integrated into your QTextEdit as follows:

```
QTextEdit *textEdit = new QTextEdit;
    textEdit->setText(
        QString::fromLatin1("This is a sample buffer. Whih this thingg will "
                 "be checkin for misstakes. Whih, Enviroment, govermant."));

    Sonnet::SpellCheckDecorator *installer =
                 new Sonnet::SpellCheckDecorator(textEdit);
    installer->highlighter()->setCurrentLanguage(QStringLiteral("en_US"));
```

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

```
class MailSpellCheckDecorator : public Sonnet::SpellCheckDecorator
{
public:
    explicit MailSpellCheckDecorator(QTextEdit *edit)
        : Sonnet::SpellCheckDecorator(edit)
    {
    }

protected:
    bool
    isSpellCheckingEnabledForBlock(const QString &blockText) const override
    {
        qDebug() << blockText;
        return !blockText.startsWith(QLatin1Char('>'));
```

```
    }
};
```

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.

8.2 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");
```

8.3 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.

```
void TestDialog::check(const QString &buffer)
{
Sonnet::Dialog *dlg = new Sonnet::Dialog(new BackgroundChecker(this), nullptr);
connect(dlg, &Dialog::spellCheckDone, this, &TestDialog::doneChecking);
dlg->setBuffer(buffer);
dlg->show();
}
```

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 9

Concurrent programming using the ThreadWeaver framework

9.1 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.

```
1  #include <QtCore>
2  #include <ThreadWeaver/ThreadWeaver>
3
4  int main(int argc, char **argv)
5  {
6      QCoreApplication app(argc, argv);
7
8      using namespace ThreadWeaver;
9      stream() << make_job([]() {
10         qDebug() << "Hello World!";
11     });
12 }
```

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.

[1] The examples are part of the ThreadWeaver source code and can be found at https://invent.kde.org/frameworks/threadweaver/-/tree/master/examples.

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 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.

9.2 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.

```
cmake_minimum_required(VERSION 3.0)
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 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.

```
find_package(KF5ThreadWeaver ${KF_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.

```
# Define the project name
project(HelloWorld)
# Add the HelloWorld executable and link the ThreadWeaver
# library to it
```

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

```
5 add_executable(ThreadWeaver_HelloWorld HelloWorld.cpp)
6 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.

Chapter 10

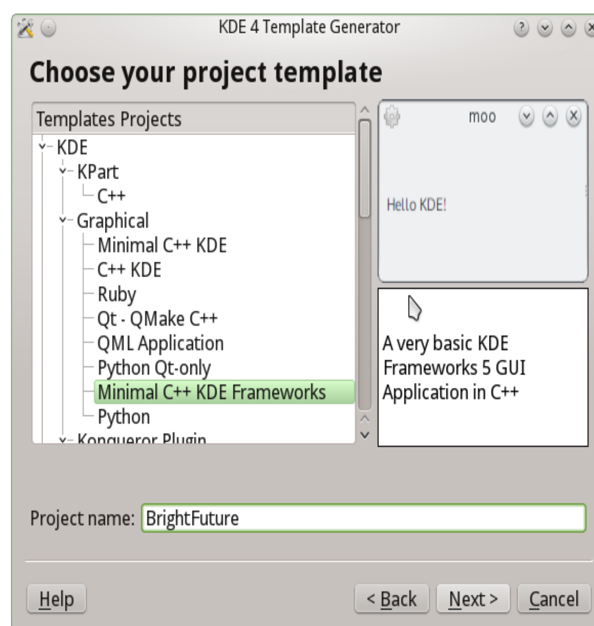
Creating a new application

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.

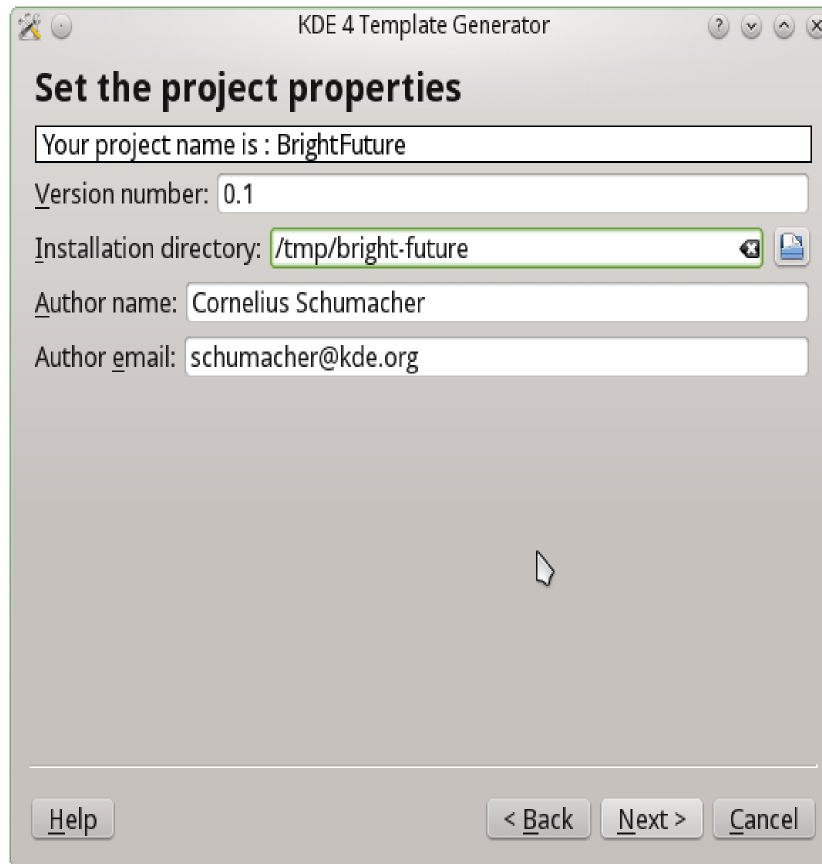
10.1 Starting a new application from a template

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.

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.

10.2 Walking through the skeleton

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

10.2.1 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:

```
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:

```
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:

```
KAboutData aboutData( QStringLiteral("brightfuture"),
            i18n("Simple App"),
            QStringLiteral("0.1"),
            18n("A Simple Application written with KDE Frameworks"),
            KAboutLicense::GPL,
            i18n("(c) 20013-2014, Cornelius Schumacher <schumacher@kde.org>"));

    aboutData.addAuthor(i18n("Cornelius Schumacher"),i18n("Author"),
                        QStringLiteral("schumacher@kde.org"));for
    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 for `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:

```
QCommandLineParser parser;
parser.addHelpOption();
parser.addVersionOption();
aboutData.setupCommandLine(&parser);
parser.process(application);
aboutData.processCommandLine(&parser)
```

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

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

10.2.2 BrightFuture

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

The header brightfuture.h is minimal:

```
/**
 * This class serves as the main window for BrightFuture.  It handles the
 * menus, toolbars and status bars.
 *
 * @short Main window class
 * @author Your Name <mail@example.com>
 * @version 0.1
 */
class BrightFuture : public QMainWindow
{
    Q_OBJECT
public:
    /**
     * Default Constructor
     */
    BrightFuture();

    /**
     * Default Destructor
     */
    virtual ~BrightFuture();

private:
    // this is the name of the root widget inside our Ui file
    // you can rename it in designer and then change it here
    Ui::mainWidget m_ui;
};
```

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:

```
1    QWidget *widget = new QWidget(this);
2    setCentralWidget(widget);
3    m_ui.setupUi(widget);
```

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

10.3 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:

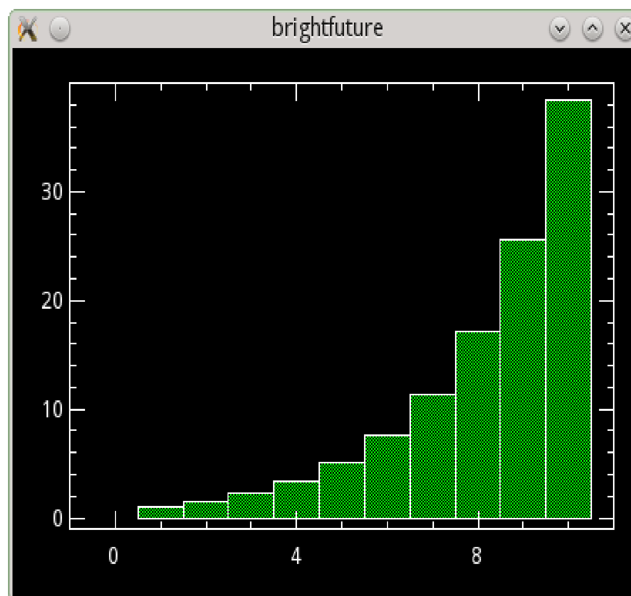
```
target_link_libraries( brightfuture
    Qt5::Widgets
    KF5::CoreAddons
    KF5::I18n
    KF5::Plotting
)
```

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:

```
1    KPlotWidget *plot = new KPlotWidget(this);
2    setCentralWidget(plot);
3
4    plot->setLimits(-1, 11, -1, 40);
5
6    KPlotObject *po =
7        new KPlotObject(Qt::white, KPlotObject::Bars, 2);
8    po->setBarBrush(QBrush(Qt::green, Qt::Dense4Pattern));
9
```

```
10      float y = 1;
11      for (float x = 1; x <= 10; x += 1) {
12          po->addPoint(x, y);
13          y *= 1.5;
14      }
15
16      plot->addPlotObject(po);
17
18      plot->update();
```

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



10.4 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.

10.4.1 Enabling KConfig

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

```
1 find_package(KF5 REQUIRED COMPONENTS
2       CoreAddons
3       I18n
4       Plotting
```

```
5       Config
6 )
```

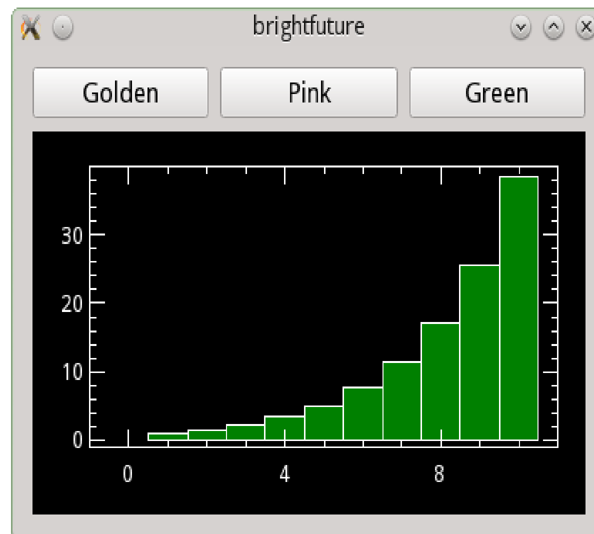
Then we need to link to the `ConfigGui` library in the CmakeList.txt file in the `src` directory to be able to access the functions `KConfig` provides:

```
1 find_package(KF5 REQUIRED COMPONENTS
2       CoreAddons
3       I18n
4       Plotting
5       Config
6 )
```

`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`.

10.4.2 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.

10.4.3 Writing the configuration

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

```
#include <KSharedConfig>
#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:

```
void BrightFuture::plotGoldenFuture()
{
    KConfigGroup config(KSharedConfig::openConfig(), "colors");
    config.writeEntry("plot", QColor("gold"));
    plotFuture();
}     KAboutData aboutData( QStringLiteral("brightfuture"),                    i18n("Simple App"),
```

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 an 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:

```
    KAboutData aboutData( QStringLiteral("brightfuture"),
            i18n("Simple App"),
            QStringLiteral("0.1"),
            i18n("A Simple Application written with KDE "
            "Frameworks"),
            KAboutLicense::GPL,
            i18n("(c) 2013-2014, "
            "Cornelius Schumacher <schumacher@kde.org>"));

```

```
10        aboutData.addAuthor(i18n("Cornelius Schumacher"),
11                            i18n("Author"),
12                            QStringLiteral("schumacher@kde.org"));
13        aboutData.setProgramIconName("brightfuture");
```

10.4.4 Reading the configuration

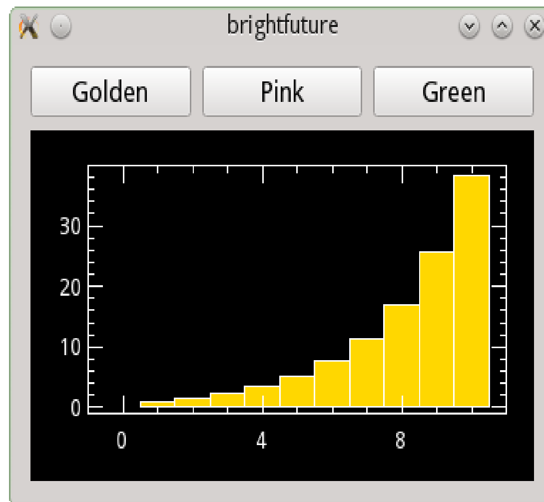
Now the final step is to read the configuration on startup of the application, so that the choice of the user is remembered.

This is done in the `plotFuture` function:

```
1 void BrightFuture::plotFuture()
2 {
3     KConfigGroup config(KSharedConfig::openConfig(), "colors");
4     QColor color = config.readEntry("plot", QColor("green"));
5     m_plot_object->setBarBrush(QBrush(color, Qt::SolidPattern));
6     m_plot->update();
7 }
```

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.

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