Qt Linguist Manual: Developers

Support for multiple languages is extremely simple in Qt applications, and adds little overhead to the developer's workload.

Qt minimizes the performance cost of using translations by translating the phrases for each window as they are created. In most applications the main window is created just once. Dialogs are often created once and then shown and hidden as required. Once the initial translation has taken place there is no further runtime overhead for the translated windows. Only those windows that are created, destroyed and subsequently created will have a translation performance cost.

Creating applications that can switch language at runtime is possible with Qt, but requires a certain amount of developer intervention and will of course incur some runtime performance cost.

Specifying Translation Sources in Qt Project Files

To enable release managers to use lupdate and lrelease, specify a `.pro` Qt project file. There must be an entry in the TRANSLATIONS section of the project file for each language that is additional to the native language. A typical entry looks like this:

```
TRANSLATIONS = arrowpad_fr.ts \
               arrowpad_nl.ts
```

Using a locale within the translation file name is useful for determining which language to load at runtime. For more information, see QLocale.

The `lupdate` tool extracts user interface strings from your application. It reads the application .pro file to identify which source files contain text to be translated. This means your source files must be listed in the SOURCES or HEADERS entry in the .pro file, or in resource files listed in the RESOURCE entry. If your files are not listed, the text in them will not be found.

An example of a complete `.pro` file with four translation source files:

```
HEADERS         = main-dlg.h \
                  options-dlg.h
SOURCES         = main-dlg.cpp \
                  options-dlg.cpp \
                  main.cpp
FORMS           = search-dlg.ui
TRANSLATIONS    = superapp_dk.ts \
```

`lupdate` expects all source code to be encoded in UTF-8 by default. Files that feature a BOM (Byte Order Mark) can also be encoded in UTF-16 or UTF-32. Set the qmake variable `CODECFORSRC` to `UTF-16` to parse files without a BOM as UTF-16.

Some editors, such as Visual Studio, however use a different encoding by default. One way to avoid encoding issues is to limit any source code to ASCII, and use escape sequences for translatable strings with other characters, for example:

```
label->setText(tr("F\374r \310lise"));
```

Internationalizing Applications

Design your application so that it can be adapted to various languages and regions without engineering changes. Qt tries to make internationalization as painless as possible for you. All input controls and text drawing methods in Qt offer built-in support for all supported languages. But you still need to keep the following things in mind when writing source code for your application:

- Make your application look for and load the appropriate translation file.
- Mark user-visible text and Ctrl keyboard accelerators as targets for translation.
- Provide context for text to be translated.
- Disambiguate identical texts.
- Use numbered arguments (%n) as placeholders for parameters that are replaced with text or numbers at run-time.
- Internationalize numbers, dates, times, and currencies.
- Mark data text strings outside functions translatable.

You can develop applications that use both C++ and QML sources in the same application and even have user interface strings in both sources. The tools create a single combined translation file and the strings are accessible from C++ and QML.

The classes that support internationalizing of Qt applications are described in Internationalization with Qt. The process of making source code translatable is described in Writing Source Code for Translation and in Internationalization and Localization with Qt Quick.

Each piece of text that requires translating requires context to help the translator identify where in the program the text appears. In the case of multiple identical texts that require different translations, the translator also requires some information to disambiguate the source texts. Marking text for translation will automatically cause the class name to be used as basic context information. In some cases the developer may be required to add additional information to help the translator.

Deploying Translations

The `.qm` files required for the application should be placed in a location where the loader code using QTranslator can locate them. Typically, this is done by specifying a path relative to QCoreApplication::applicationDirPath().

In Qt 4, there is one big, monolithic `.qm` file per locale. For example, the file `qt_de.qm` contains the German translation of all libraries.

In Qt 5, the `.qm` files were split up by module and there is a so-called meta catalog file which includes the `.qm` files of all modules. The name of the meta catalog file is identical to the name of Qt 4's monolithic `.qm` file so that existing loader code works as before provided all included `.qm` files are found.

However, it is not necessary to always deploy all of Qt 5's `.qm` files. We recommend concatenating the `.qm` files of the modules required to one file matching the meta catalog file name using the tool `lconvert` in the deploy step. For example, to create a German translation file for an application using the modules Qt Core, Qt GUI, and Qt Quick, run:

```
lconvert -o installation_folder/qt_de.qm qtbase_de.qm qtdeclarative_de.qm
```

Tutorials

The following tutorials illustrate how to prepare Qt applications for translation:

- Hello tr() is a C++ application that demonstrates the creation of a QTranslator object. It also shows the simplest use of the `tr()` function to mark user-visible source text for translation.
- Arrow Pad is a C++ application that demonstrates how to make the application load the translation file applicable to the current locale. It also shows the use of the two-argument form of `tr()` which provides additional information to the translator.
- Troll Print is a C++ application that demonstrates how identical source texts can be distinguished even when they occur in the same context. This tutorial also discusses how the translation tools help minimize the translator's work when an application is upgraded.
- Internationalization is a Qt Quick application that demonstrates how to internationalize Qt Quick applications.

‹ Qt Linguist Manual: Translators

Qt Linguist Manual: TS File Format ›