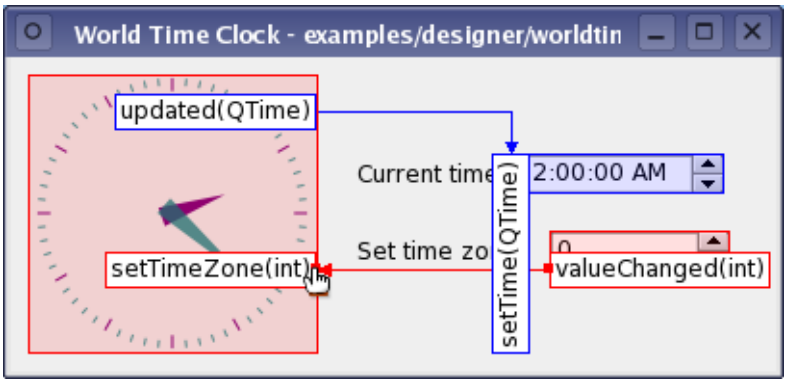
Creating Custom Widgets for Qt Designer

Qt Designer's plugin-based architecture allows user-defined and third party custom widgets to be edited just like you do with standard Qt widgets. All of the custom widget's features are made available to *Qt Designer*, including widget properties, signals, and slots. Since *Qt Designer* uses real widgets during the form design process, custom widgets will appear the same as they do when previewed.



The QtDesigner module provides you with the ability to create custom widgets in *Qt Designer*.

Getting Started

To integrate a custom widget with *Qt Designer*, you require a suitable description for the widget and an appropriate `.pro` file.

Providing an Interface Description

To inform *Qt Designer* about the type of widget you want to provide, create a subclass of QDesignerCustomWidgetInterface that describes the various properties your widget exposes. Most of these are supplied by functions that are pure virtual in the base class, because only the author of the plugin can provide this information.

Function	Description of the return value
`name()`	The name of the class that provides the widget.
`group()`	The group in *Qt Designer*'s widget box that the widget belongs to.
`toolTip()`	A short description to help users identify the widget in *Qt Designer*.
`whatsThis()`	A longer description of the widget for users of *Qt Designer*.
`includeFile()`	The header file that must be included in applications that use this widget. This information is stored in UI files and will be used by `uic` to create a suitable `#includes` statement in the code it generates for the form containing the custom widget.
`icon()`	An icon that can be used to represent the widget in *Qt Designer*'s widget box.

Function Description of the return value

createWidget()	A QWidget pointer to an instance of the custom widget, constructed with the parent supplied. **Note:** createWidget() is a factory function responsible for creating the widget only. The custom widget's properties will not be available until load() returns.
domXml()	A description of the widget's properties, such as its object name, size hint, and other standard QWidget properties.
codeTemplate()	This function is reserved for future use by *Qt Designer*.

Two other virtual functions can also be reimplemented:

initialize()	Sets up extensions and other features for custom widgets. Custom container extensions (see QDesignerContainerExtension) and task menu extensions (see QDesignerTaskMenuExtension) should be set up in this function.
isInitialized()	Returns true if the widget has been initialized; returns false otherwise. Reimplementations usually check whether the initialize() function has been called and return the result of this test.

Notes on the domXml() Function

The domXml() function returns a UI file snippet that is used by *Qt Designer*'s widget factory to create a custom widget and its applicable properties.

Since Qt 4.4, *Qt Designer*'s widget box allows for a complete UI file to describe **one** custom widget. The UI file can be loaded using the <ui> tag. Specifying the <ui> tag allows for adding the <customwidget> element that contains additional information for custom widgets. The <widget> tag is sufficient if no additional information is required

If the custom widget does not provide a reasonable size hint, it is necessary to specify a default geometry in the string returned by the domXml() function in your subclass. For example, the AnalogClockPlugin provided by the Custom Widget Plugin example, defines a default widgetgeometry in the following way:

```
    ...
R"(
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>100</width>
        <height>100</height>
      </rect>
    </property>
")
    ...
```

An additional feature of the domXml() function is that, if it returns an empty string, the widget will not be installed in *Qt Designer*'s widget box. However, it can still be used by other widgets in the form. This feature is used to hide widgets that should not be explicitly created by the user, but are required by other widgets.

A complete custom widget specification looks like:

```
<ui language="c++"> displayname="MyWidget">
```

```
            <customwidget>
                <class>widgets::MyWidget</class>
                <addpagemethod>addPage</addpagemethod>
                <propertyspecifications>
                    <stringpropertyspecification name="fileName" notr="true" type="singleline"/>
                    <stringpropertyspecification name="text" type="richtext"/>
                    <tooltip name="text">Explanatory text to be shown in Property Editor</tooltip>
                </propertyspecifications>
            </customwidget>
        </customwidgets>
    </ui>
```

Attributes of the <ui> tag:

Attribute	Presence	Values	Comment
language	optional	"c++", "jambi"	This attribute specifies the language the custom widget is intended for. It is mainly there to prevent C++-plugins from appearing in Qt Jambi.
displayname	optional	Class name	The value of the attribute appears in the Widget box and can be used to strip away namespaces.

The <addpagemethod> tag tells *Qt Designer* and uic which method should be used to add pages to a container widget. This applies to container widgets that require calling a particular method to add a child rather than adding the child by passing the parent. In particular, this is relevant for containers that are not a a subclass of the containers provided in *Qt Designer*, but are based on the notion of *Current Page*. In addition, you need to provide a container extension for them.

The <propertyspecifications> element can contain a list of property meta information.

The tag <tooltip> may be used to specify a tool tip to be shown in Property Editor when hovering over the property. The property name is given in the attribute name and the element text is the tooltip. This functionality was added in Qt 5.6.

For properties of type string, the <stringpropertyspecification> tag can be used. This tag has the following attributes:

Attribute	Presence	Values	Comment
name	required	Name of the property	
type	required	See below table	The value of the attribute determines how the property editor will handle them.
notr	optional	"true", "false"	If the attribute is "true", the value is not meant to be translated.

Values of the type attribute of the string property:

Value	Type
"richtext"	Rich text.
"multiline"	Multi-line plain text.
"singleline"	Single-line plain text.
"stylesheet"	A CSS-style sheet.
"objectname"	An object name (restricted set of valid characters).
"url"	URL. file name.

Plugin Requirements

In order for plugins to work correctly on all platforms, you need to ensure that they export the symbols needed by *Qt Designer*.

First of all, the plugin class must be exported in order for the plugin to be loaded by *Qt Designer*. Use the Q_PLUGIN_METADATA() macro to do this. Also, the QDESIGNER_WIDGET_EXPORT macro must be used to define each custom widget class within a plugin, that *Qt Designer* will instantiate.

Creating Well Behaved Widgets

Some custom widgets have special user interface features that may make them behave differently to many of the standard widgets found in *Qt Designer*. Specifically, if a custom widget grabs the keyboard as a result of a call to QWidget::grabKeyboard(), the operation of *Qt Designer* will be affected.

To give custom widgets special behavior in *Qt Designer*, provide an implementation of the initialize() function to configure the widget construction process for *Qt Designer* specific behavior. This function will be called for the first time before any calls to createWidget() and could perhaps set an internal flag that can be tested later when *Qt Designer* calls the plugin's createWidget() function.

Building and Installing the Plugin

A Simple Plugin

The Custom Widget Plugin Example demonstrates a simple *Qt Designer* plugin.

The `.pro` file for a plugin must specify the headers and sources for both the custom widget and the plugin interface. Typically, this file only has to specify that the plugin's project is to be built as a library, but with specific plugin support for *Qt Designer*. This is done with the following declarations:

```
QT          += widgets uiplugin
CONFIG      += plugin
TEMPLATE    = lib
```

The QT variable contains the keyword `uiplugin`. It indicates that the plugin uses the abstract interfaces QDesignerCustomWidgetInterface and QDesignerCustomWidgetCollectionInterface only and has no linkage to the *Qt Designer* libraries. When accessing other interfaces of *Qt Designer* that have linkage, `designer` should be used instead; this ensures that the plugin dynamically links to the *Qt Designer* libraries and has a run-time dependency on them.

If plugins are built in a mode that is incompatible with *Qt Designer*, they will not be loaded and installed. For more information about plugins, see the Plugins HOWTO document.

It is also necessary to ensure that the plugin is installed together with other *Qt Designer* widget plugins:

```
target.path = $$[QT_INSTALL_PLUGINS]/designer
INSTALLS += target
```

The `$[QT_INSTALL_PLUGINS]` variable is a placeholder to the location of the installed Qt plugins. You can configure *Qt Designer* to look for plugins in other locations by setting the `QT_PLUGIN_PATH` environment variable before running the application.

See QCoreApplication::libraryPaths() for more information about customizing paths for libraries and plugins with Qt applications.

Splitting up the Plugin

In a real world scenario, you do not want to have dependencies of the application making use of the custom widgets to the *Qt Designer* headers and libraries as introduced by the simple approach explained above.

The following sections describe how to resolve this.

Linking the Widget into the Application

The source and header file of the custom widget can be shared between the application and *Qt Designer* by creating a `.pri` file for inclusion:

```
INCLUDEPATH += $$PWD
HEADERS += $$PWD/analogclock.h
SOURCES += $$PWD/analogclock.cpp
```

This file would then be included by the `.pro` file of the plugin and the application:

```
include(customwidget.pri)
```

Sharing the Widget Using a Library

Another approach is to put the widget into a library that is linked to the *Qt Designer* plugin as well as to the application. It is recommended to use static libraries to avoid problems locating the library at run-time.

For shared libraries, see Creating Shared Libraries.

Using the Plugin with QUiLoader

The preferred way of adding custom widgets to QUiLoader is to subclass it reimplementing QUiLoader::createWidget().

However, it is also possible to use *Qt Designer* custom widget plugins (see QUiLoader::pluginPaths() and related functions). To avoid having to deploy the *Qt Designer* libraries onto the target device, those plugins should have no linkage to the *Qt Designer* libraries (`QT = uiplugin`, see Creating Custom Widgets for Qt Designer#BuildingandInstallingthePlugin).

Related Examples

For more information on using custom widgets in *Qt Designer*, refer to the Custom Widget Plugin and World Time Clock Plugin examples for more information about using custom widgets in *Qt Designer*. Also, you can use the QDesignerCustomWidgetCollectionInterface class to combine several custom widgets into a single library.

‹ Using Custom Widgets with Qt Designer

Creating Custom Widget Extensions ›