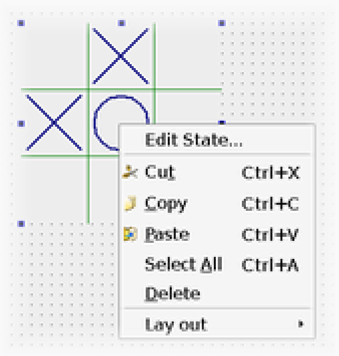
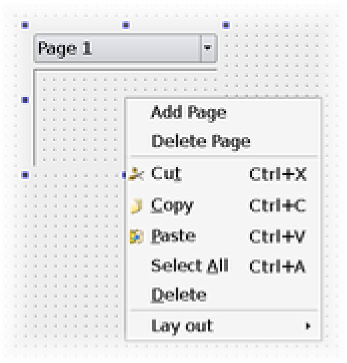
Creating Custom Widget Extensions

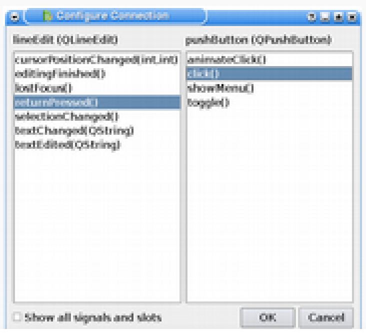
Once you have a custom widget plugin for *Qt Designer*, you can provide it with the expected behavior and functionality within *Qt Designer*'s workspace, using custom widget extensions.

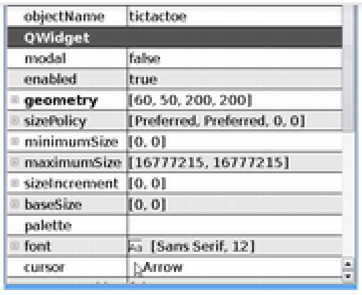
Extension Types

There are several available types of extensions in *Qt Designer*. You can use all of these extensions in the same pattern, only replacing the respective extension base class.

QDesignerContainerExtension is necessary when implementing a custom multi-page container.

 Edit State... Cut Ctrl+X Copy Ctrl+C Paste Ctrl+V Select All Ctrl+A Delete Lay out	**QDesignerTaskMenuExtension** QDesignerTaskMenuExtension is useful for custom widgets. It provides an extension that allows you to add custom menu entries to *Qt Designer*'s task menu. The Task Menu Extension example illustrates how to use this class.
 Page 1 Add Page Delete Page Cut Ctrl+X Copy Ctrl+C Paste Ctrl+V Select All Ctrl+A Delete Lay out	**QDesignerContainerExtension** QDesignerContainerExtension is necessary when implementing a custom multi-page container. It provides an extension that allows you to add and delete pages for a multi-page container plugin in *Qt Designer*. The Container Extension example further explains how to use this class. **Note:** It is not possible to add custom per-page properties for some widgets (e.g., QTabWidget) due to the way they are implemented.

 Configure Connection
lineEdit (QLineEdit)
pushButton (QPushButton)
cursorPositionChanged(int,int)
editingFinished()
lostFocus()
returnPressed()
selectionChanged()
textChanged(QString)
textEdited(QString)
animateClick()
click()
showMenu()
toggle()
Show all signals and slots
OK
Cancel
	QDesignerMemberSheetExtension
	The QDesignerMemberSheetExtension class allows you to manipulate a widget's member functions displayed when connecting signals and slots.



objectName	tictactoe
QWidget	
modal	false
enabled	true
geometry	[60, 50, 200, 200]
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	[0, 0]
maximumSize	[16777215, 16777215]
sizeIncrement	[0, 0]
baseSize	[0, 0]
palette	
font	[Sans Serif, 12]
cursor	Arrow

displayed in *Qt Designer*'s property editor.

Qt Designer uses the QDesignerPropertySheetExtension and the QDesignerMemberSheetExtension classes to feed its property and signal and slot editors. Whenever a widget is selected in its workspace, *Qt Designer* will query for the widget's property sheet extension; likewise, whenever a connection between two widgets is requested, *Qt Designer* will query for the widgets' member sheet extensions.

Warning: All widgets have default property and member sheets. If you implement custom property sheet or member sheet extensions, your custom extensions will override the default sheets.

Creating an Extension

To create an extension you must inherit both QObject and the appropriate base class, and reimplement its functions. Since we are implementing an interface, we must ensure that it is made known to the meta object system using the Q_INTERFACES() macro in the extension class's definition. For example:

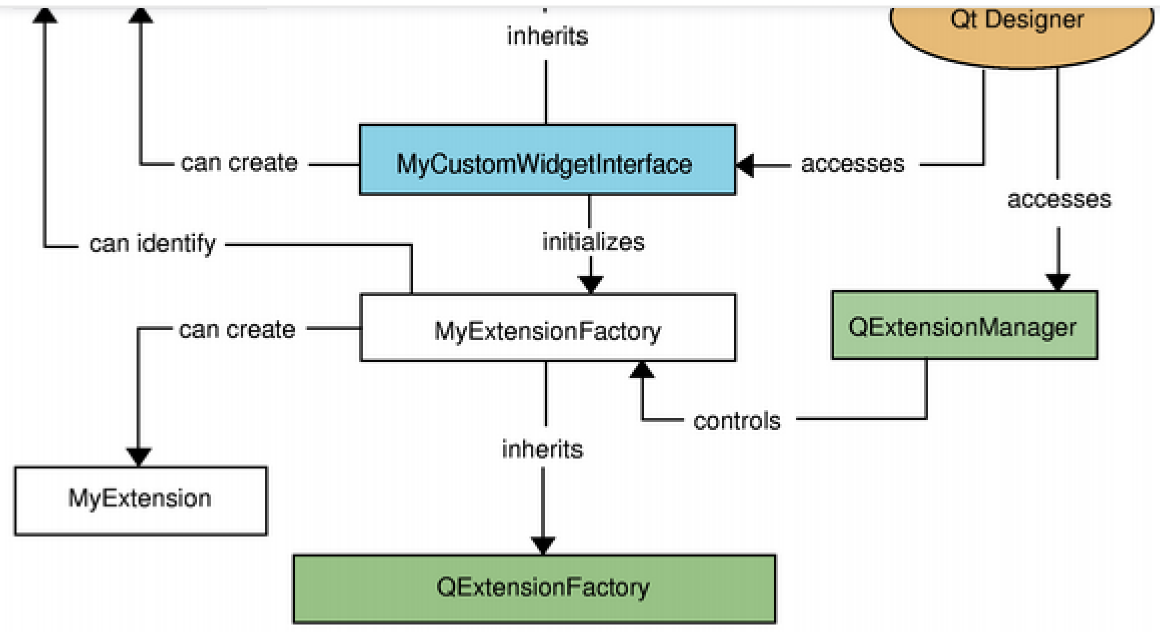
```
class MyExtension: public QObject,
                   public QdesignerContainerExtension
{
    Q_OBJECT
    Q_INTERFACE(QDesignerContainerExtension)

    ...
}
```

This enables *Qt Designer* to use the qobject_cast() function to query for supported interfaces using a QObject pointer only.

Exposing an Extension to Qt Designer

In *Qt Designer* the extensions are not created until they are required. For this reason, when implementing extensions, you must subclass QExtensionFactory to create a class that is able to make instances of your extensions. Also, you must register your factory with *Qt Designer*'s extension manager; the extension manager handles the construction of extensions.

When an extension is requested, *Qt Designer*'s extension manager will run through its registered factories calling QExtensionFactory::createExtension() for each of them until it finds one that is able to create the requested extension for the selected widget. This factory will then make an instance of the extension.



Creating an Extension Factory

The QExtensionFactory class provides a standard extension factory, but it can also be used as an interface for custom extension factories.

The purpose is to reimplement the QExtensionFactory::createExtension() function, making it able to create your extension, such as a MultiPageWidget container extension.

You can either create a new QExtensionFactory and reimplement the QExtensionFactory::createExtension() function:

```
QObject *ANewExtensionFactory::createExtension(QObject *object,
        const QString &iid, QObject *parent) const
{
    if (iid != Q_TYPEID(QDesignerContainerExtension))
        return 0;

    if (MyCustomWidget *widget = qobject_cast<MyCustomWidget*>
            (object))
        return new MyContainerExtension(widget, parent);

    return 0;
}
```

or you can use an existing factory, expanding the QExtensionFactory::createExtension() function to enable the factory to create your custom extension as well:

```
QObject *AGeneralExtensionFactory::createExtension(QObject *object,
        const QString &iid, QObject *parent) const
{
    MyCustomWidget *widget = qobject_cast<MyCustomWidget*>(object);

    if (widget && (iid == Q_TYPEID(QDesignerTaskMenuExtension))) {
            return new MyContainerExtension(widget, parent);

        } else {
            return 0;
        }
    }
```

Accessing Qt Designer's Extension Manager

When implementing a custom widget plugin, you must subclass the QDesignerCustomWidgetInterface to expose your plugin to *Qt Designer*. This is covered in more detail in the Creating Custom Widgets for Qt Designer section. The registration of an extension factory is typically made in the QDesignerCustomWidgetInterface::initialize() function:

```
void MyPlugin::initialize(QDesignerFormEditorInterface *formEditor)
{
    if (initialized)
        return;

    QExtensionManager *manager = formEditor->extensionManager();
    Q_ASSERT(manager != 0);

    manager->registerExtensions(new MyExtensionFactory(manager),
                                Q_TYPEID(QDesignerTaskMenuExtension));

    initialized = true;
}
```

The `formEditor` parameter in the QDesignerCustomWidgetInterface::initialize() function is a pointer to *Qt Designer*'s current QDesignerFormEditorInterface object. You must use the QDesignerFormEditorInterface::extensionManager() function to retrieve an interface to *Qt Designer*'s extension manager. Then you use the QExtensionManager::registerExtensions() function to register your custom extension factory.

Related Examples

For more information on creating custom widget extensions in *Qt Designer*, refer to the Task Menu Extension and Container Extension examples.

< Creating Custom Widgets for Qt Designer Qt Designer's UI File Format >