

# Creating KParts components, Part 1: Build KParts components

Skill Level: Introductory

[David Faure \(faure@kde.org\)](mailto:faure@kde.org)  
Maintainer of Konqueror  
MandrakeSoft

21 May 2002

This tutorial shows you how to build and deploy a KParts component, how to provide extra functionality through the component's actions, and how to improve integration into Konqueror with the Browser Extension.

## Section 1. About this tutorial

This tutorial shows you how to create KParts components. It introduces the core KParts concepts of read-only and read-write parts and network transparency.

KParts is the component technology that has been introduced in KDE 2. KParts allows KDE applications that need the same functionality to share a component for this task, "embedding" the graphical component into the application's window. KParts components are mainly viewers and editors.

This tutorial presents the classes provided by the KParts framework: `ReadOnlyPart` for viewers, and `ReadWritePart` for editors. The tutorial then demonstrates how to create a basic component. The example component can view XML files as a tree of tags. This example uses the `QListView` widget (included in the Qt library).

The tutorial then shows how to deploy the component so that Konqueror can use it, and later the tutorial shows how to add actions to the component, including the XML-GUI file to position those actions in the GUI. Lastly, it demonstrates how to transform a read-only part into a read-write part.

The second KParts tutorial, "[Creating KParts components, Part 2: Use KParts components in a KDE application](#)," shows how to write KDE applications that are able to utilize KParts components, either those provided by KDE or custom ones.

## Prerequisites

Readers should be familiar with C++ application development: classes, methods, members, etc. It's also helpful to have a basic knowledge about Qt, such as signals and slots. Previous experience with KDE development will help in understanding the tutorial, although it isn't required.

To gain an understanding of the KParts component architecture, read the article "[Coding with KParts](#)," also by David Faure.

To run the code in this tutorial, you need the following tools:

- KDE 2.x or 3.x and its development packages. KDE 3 is recommended, since it simplifies some of the code needed to use KParts. [Download KDE](#) from kde.org.
- A C++ compiler (usually gcc) and other standard GNU programming tools (make, autoconf etc.), which all come with any Linux distribution. You can [download gcc](#) from GNU.
- Developers who prefer an IDE instead of a simple text editor can use [KDevelop](#).
- Without `KDevelop`, it is necessary to use the `kdesdk` package to generate a compilation framework. You can [get kdesdk](#) from kde.org (as well as from many other places Web-wide).

## Technical terms

- The tutorial uses the words *component* and *part* interchangeably, since a KParts component is called a *part*.
- The variable `$prefix` designates the prefix (in other words, base directory) where KDE is installed. This is `/usr` on many distributions. The best way to check is to run `kde-config --prefix`.

---

## Section 2. Read-only and read-write parts

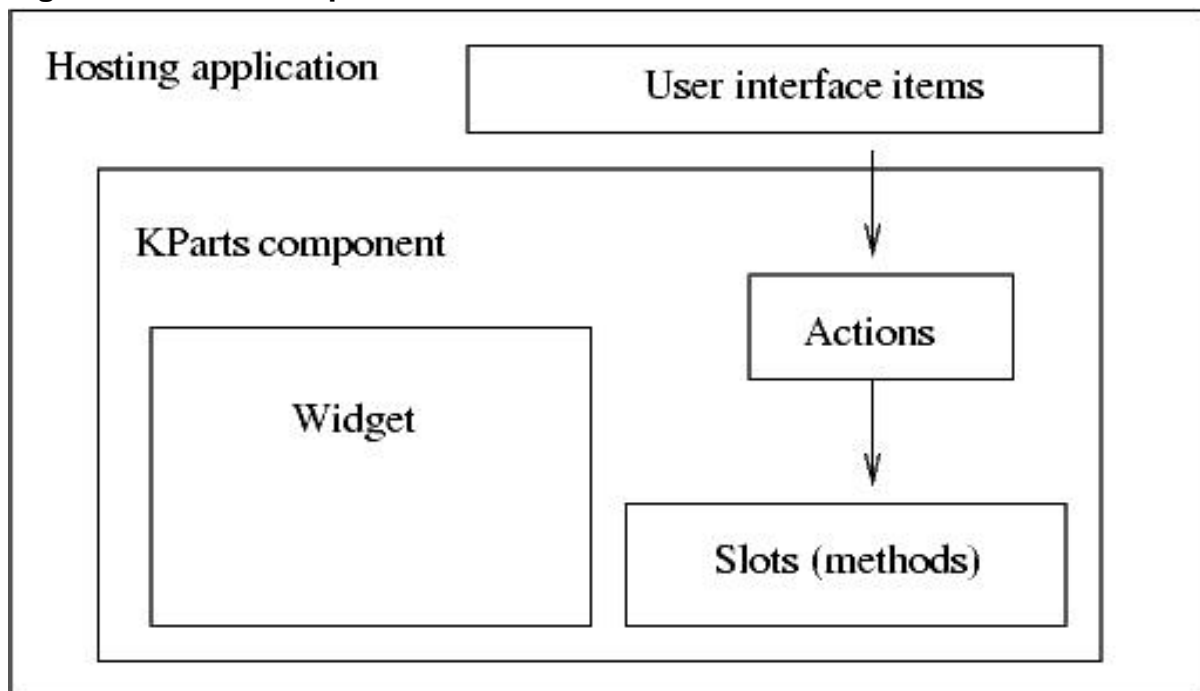
## What makes a KParts component?

Before creating a KParts component, it is important to understand what the KParts framework provides (i.e., network transparency) and what it requires from components. This knowledge enables a developer to choose the right class from which to inherit.

Each KParts component must be made up of three elements:

- A widget (must be a descendant of `QWidget`)
- The functionality that is provided by the component, in addition to the widget
- A user interface to access that functionality (actions, and XML file)

**Figure 1. KParts component structure**



For instance, a text editor KParts component consists of:

- A multiline edit widget
- Additional functionality such as Find, Replace, SpellCheck, Wordwrap settings, etc.
- The menu items and toolbar buttons for the widget's functionality

The application using (or "hosting") the component embeds the component's widget and merges the component's user interface with its own in order to provide access to the component's complete functionality.

## Read-Only and Read-Write components

There are two basic types of KParts components: Read-Only and Read-Write.

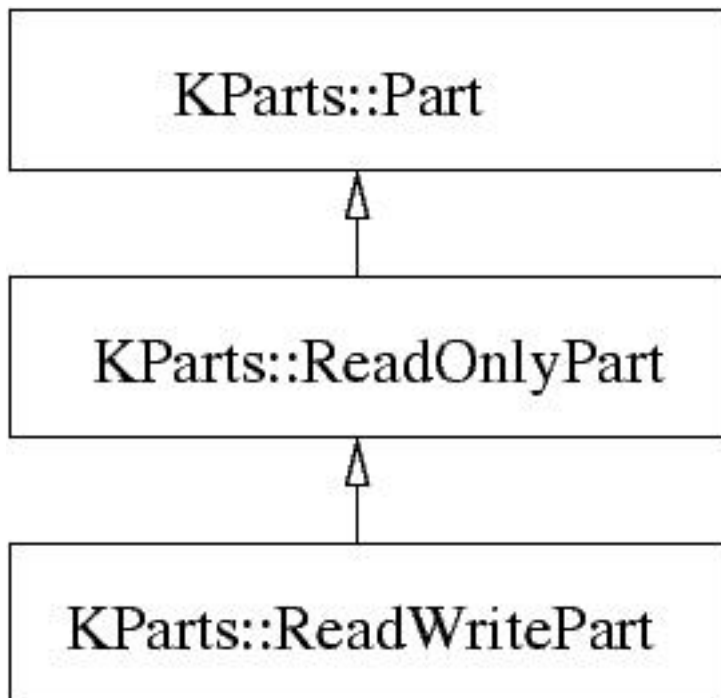
### Read-Only components

The `ReadOnlyPart` class provides a common framework for all parts that implement any kind of viewer. Some examples of viewers are: a text viewer, an image viewer, a PostScript viewer and a Web browser. All of these viewers can display a file, given its URL, and prevent any modification to the file.

### Read-Write components

The `ReadWritePart` class, which is an extension of `ReadOnlyPart`, adds the possibility of modifying and saving the document. This class is used by text editor components in KDE. Note that any read-write component must implement a read-only mode, in which it disguises itself as a `ReadOnlyPart`, so that it can be used as a viewer.

**Figure 2. KParts class hierarchy**



The KParts classes are defined in the `KParts` namespace, which explains the

naming of `KParts::ReadOnlyPart` and `KParts::ReadWritePart`.

## Network transparency provided by the KParts framework

Network transparency allows the user of an application to work on remote files in the same way as he or she works on local files. KDE's file selector is able to list remote directories, for instance, over FTP, SMB, and even Webdav (in KDE3). All KParts components, like most KDE applications, can seamlessly read and write remote files. It has always been a design decision in KDE to provide network transparency wherever possible, which is why most KDE applications use the URL instead of the filename to describe the way to access a document.

A read-only part acts on a URL in a read-only way, usually to display it. The KParts framework defines methods for opening a URL, for closing a URL, and above all it provides network transparency by downloading the file, if it is remote, and emitting signals (started, progression, completed) so that the hosting application can show progress information. The important point is that the part itself has to provide only `openFile()`, which opens a local file.

Similarly, a read-write part has to provide only `saveFile()`. The KParts framework provides the necessary network transparency, i.e. uploading the file after it has been modified. The framework also takes care of the "modified" status of the document; the part simply calls `setModified()` when necessary, and the framework prompts the user for saving when closing a modified document, etc.

Note that it is possible to override this default behavior by implementing the virtual methods `openURL()` and `closeURL()` instead of `openFile()` and `saveFile()`.

Refer to the API documentation for KParts for more information about those methods (See Resources for a link).

This section introduced the main classes provided by KParts: `ReadOnlyPart`, `ReadWritePart`. In the rest of this tutorial, we will first create a read-only component, and then modify it to be a read-write component.

---

## Section 3. Writing a basic KParts component

### Skeleton for a read-only part

This section shows how to write a read-only KParts component that provides some

functionality, and the necessary code to make it possible to create the component from the hosting application. The next section will explain how to deploy and test the component.

The following example is a part, called `KXMLTreePart`, that shows an XML file as a tree of tags. Since it does not allow data to be modified, it inherits `KParts::ReadOnlyPart`. The widget it uses is `QListView`, the Qt widget that implements both flat lists and trees, with a multi-columns ability.

```
#ifndef __kxmltreepart_h__
#define __kxmltreepart_h__

#include <kparts/part.h>

// Forward declarations, saves time compared to including the full headers
class QListView;
class KAboutData;

class KXMLTreePart : public KParts::ReadOnlyPart
{
    Q_OBJECT
public:
    KXMLTreePart( QWidget *parentWidget, const char *widgetName,
                 QObject *parent, const char *name, const QStringList &args );
    virtual ~KXMLTreePart() {}

    // Return information about the part
    static KAboutData* createAboutData();

protected:
    // Open the file whose path is stored in the member variable m_file
    // and return true on success, false on failure.
    virtual bool openFile();

private:
    // The component's widget
    QListView *m_listView;
};

#endif
```

## The part constructor

The most important thing in the definition of a KParts component is its constructor. To allow the component to have its own "identity" and not the one of the hosting application, it must create its own `KInstance`. This allows the component to access its own data files, saved under `$prefix/share/apps/instancename`.

The constructor must at least define the instance and the widget. If the component provides actions, it must create them in the constructor and define the corresponding XML File. In short, the code for any KParts constructor must at least call `setInstance()`, `setWidget()`, and `setXMLFile()`.

```

KXMLTreePart::KXMLTreePart( QWidget *parentWidget, const char *widgetName,
    QObject *parent, const char *name, const QStringList &)
: KParts::ReadOnlyPart( parent, name )
{
    setInstance( KXMLTreeFactory::instance() );

    // Create the tree widget, and set it as the part's widget
    m_listView = new QListView( parentWidget, widgetName );
    setWidget( m_listView );
    // Ask for one column in the tree and hide the horizontal header at the top
    m_listView->addColumn("Text");
    m_listView->header()->hide();
    // By default the root items don't display a '+' sign in QListView
    m_listView->setRootIsDecorated(true);

    // Action creation code will go here
    //setXMLFile( "kxmltreepart.rc" );
}

```

The five-arguments signature for the constructor is mandatory: it allows the application to pass a parent widget for the part's widget, which is different from the parent object for the part itself. The last argument is a list of additional parameters the application can pass to the part. For example, this allows several parts to be defined in the same library.

## Libraries and factories

Before finishing the code for the `KXMLTreePart` component, it is necessary to understand what factories are and how shared libraries are used by KParts.

Reusability of KParts components implies that components be accessible to all KDE applications. The only way to do that is to compile KParts component in a shared library. Of course it is possible to make a KParts component statically linked to application code but then this component isn't available to other applications.

An application willing to dynamically open a shared library uses the `KLibLoader` class. `KLibLoader` takes care of locating the library, opening it, and calling an initialization function, which is the entry point into the library.

This initialization function creates a factory, which always inherits `KLibFactory`, the base class for all factories. The factory is the class that creates the object located in the library, which is the KParts component in this case. `KLibFactory` provides a virtual method, which the component's factory implements to create the component.

## Factory code to create the part

Factory code is usually "boiler-plate" code, and is the same for all parts.

KDE 3 provides macros and templates that reduce the amount of code needed to create a factory to two lines. The first line defines the factory for the `KXMLTreePart`, and the second line makes the factory externally available, in the `libkxmltreepart` library.

```
#include <kparts/genericfactory.h>
// Factory code for KDE 3
typedef KParts::GenericFactory<KXMLTreePart> KXMLTreeFactory;
K_EXPORT_COMPONENT_FACTORY( libkxmltreepart /*library name*/, KXMLTreeFactory );
```

## KDE 2 code to create a factory

KDE 2 requires much more code to create a factory. Instead of the two lines above, you must include `kxmltreepart_factory.cpp`, which contains the code below.

```
#include "kxmltreepart.h"
#include <kparts/factory.h>
#include <kinstance.h>
#include <kaboutdata.h>

class KXMLTreeFactory : public KParts::Factory
{
public:
    KXMLTreeFactory() : KParts::Factory() {}
    virtual ~KXMLTreeFactory() {
        delete s_instance;
        delete s_about;
        s_instance = 0L;
    }

    virtual KParts::Part *createPartObject(
        QWidget *parentWidget = 0, const char *widgetName = 0,
        QObject *parent = 0, const char *name = 0,
        const char *classname = "KParts::Part",
        const QStringList &args = QStringList() )
    {
        KXMLTreePart* obj = new KXMLTreePart( parentWidget, widgetName,
                                             parent, name, args );
        // Code for read-write parts: set read-write status depending on classname
        //if (QString(classname) == "KParts::ReadWritePart")
        //    obj->setReadWrite(true);
        // otherwise, it has to be readonly
        //assert (QString(classname) == "KParts::ReadOnlyPart");
        return obj;
    }

    static KInstance* instance() {
        if( !s_instance )
        {
            s_about = KXMLTreePart::createAboutData();
            s_instance = new KInstance( s_about );
        }
        return s_instance;
    }
private:
    static KInstance* s_instance;
    static KAboutData* s_about;
};
```

```

KInstance* KXMLTreeFactory::s_instance = 0L;
KAboutData* KXMLTreeFactory::s_about = 0L;

extern "C"
{
    void* init_libkxmltreepart() {
        return new KXMLTreeFactory;
    }
};

```

The code generated by the `KParts::GenericFactory` template in KDE3 is the same as the code for the factory that must be written manually in the KDE 2 version.

The `K_EXPORT_COMPONENT_FACTORY` macro generates the `extern "C"` symbol definition, which is the entry point for the library. It needs to be linked as a C function to avoid C++ name mangling. C linkage means that the symbol in the library will match the function name.

## Defining the "about data" for the part

In both cases (KDE 2 and KDE 3), the factory previously defined needs `KXMLTreePart::createAboutData()`, a static method in which `KXMLTreePart` defines data about the part: its instance name, the name that appears to the user, version, license, and author information.

```

KAboutData* KXMLTreePart::createAboutData()
{
    KAboutData* aboutData =
        new KAboutData( "kxmltreepart", I18N_NOOP("KXmlTreePart"),
            "0.1", I18N_NOOP( "XML Tree Viewer" ),
            KAboutData::License_GPL,
            "(c) 2001, David Faure <david@mandrakesoft.com>");
    return aboutData;
}

```

`I18N_NOOP` is a macro that allows translation of the string to be delayed until actually displaying the string to the user. The usual `i18n()` translates immediately, but the `KInstance` needs to exist already.

Much more information can be added in the `KAboutData` object, such as authors, home page, and bug-report address. See the API documentation for details.

## Implementing the `openFile()` method

The `KXMLTreePart` component is almost ready, it simply needs to display something. For this it needs to implement `openFile()`, to read the contents of the file whose path is stored in the `m_file` member variable.

```
bool KXMLTreePart::openFile()
{
    kdDebug() << "KXMLTreePart: opening " << m_file << endl;
    QDomDocument doc;
    QFile file( m_file ); // Open the file
    doc.setContent( &file ); // Parse the file as a DOM document
    QDomElement elem = doc.documentElement(); // Get the root element
    if ( !elem.isNull() )
    {
        QListViewItem* rootItem = new QListViewItem( m_listView, 0L, elem.tagName() );
        fillTreeView( elem, rootItem );
    }
    // Open the root item
    if (m_listView->firstChild()) m_listView->firstChild()->setOpen(true);
    // Some feedback for the user
    emit statusBarText( i18n("Parsing complete.") );
    return true;
}

void KXMLTreePart::fillTreeView( QDomElement& parentElem, QListViewItem* parent )
{
    QDomElement elem = parentElem.firstChild().toElement();
    QListViewItem *lastItem = 0L;
    for ( ; !elem.isNull(); elem = elem.nextSibling().toElement() ) {
        lastItem = new QListViewItem( parent, lastItem, elem.tagName() );
        fillTreeView( elem, lastItem );
    }
}
```

`fillTreeView` is the recursive method that creates the items in the tree for all children of a DOM element. Don't forget to declare it (protected) in the header file.

For more details, read the API documentation for `QListViewItem` and `QDomDocument`, this is specific to this example and unrelated to the general case.

---

## Section 4. Deploying a KParts component

### Creating a Makefile to build the library

Even though the code for the `KXMLTreePart` component is now ready, the component isn't yet ready to be used by other applications. To accomplish this the code needs to be compiled as a shared library, a `.desktop` file describing the component must be written, and both need to be installed to the appropriate location.

The first step is to compile the component as a shared library, which is more easily done with a `Makefile.am`, processed by `automake`, than by writing a `Makefile` by hand. There are three ways of generating the framework for compiling a KDE application:

- Using `KAppTemplate`, part of the `kdesdk` package
- Using `KDevelop`, the all-in-one IDE
- Using an existing KDE source module -- for instance, using `kdelibs/kparts/tests` for saving the component's files and compiling it there

## The code for the `Makefile.am` file

Here is what the `Makefile.am` should contain:

```
INCLUDES = $(all_includes)

# These are not really libraries, but modules dynamically opened.
# So they should be installed in kde_module_dir, which is usually $prefix/lib/kde3
kde_module_LTLIBRARIES = libkxmltreepart.la

libkxmltreepart_la_SOURCES = kxmltreepart.cpp
libkxmltreepart_la_LIBADD = $(LIB_KPARTS)
libkxmltreepart_la_LDFLAGS = $(all_libraries) $(KDE_PLUGIN)

# Automatically generate moc files
METASOURCES = AUTO

# Install the .desktop file into the kde_services directory
kde_services_DATA = kxmltreepart.desktop
```

This should be sufficient to compile the code and create the library.

## Desktop files: describing applications and services

But making a shared library available is not enough. Applications need to know about the library's existence and what it provides. In KDE this is realized by `.desktop` files.

`.desktop` files describe the applications installed on the system. Named `.kdelnk` in KDE 1, they have been renamed `.desktop` in KDE 2 in order to follow the Desktop Entry Standard. Their syntax is quite standard: lists of `Key=Value` pairs, one per line, organized in groups.

The files describing applications contain the line `Type=Application`, and the name, icon, comment, and execution command relating to the application. They are installed under the `share/applnk` directory, and form the "K menu".

`.desktop` files describing libraries are known as *service* desktop files and contain the line `Type=Service`. They are installed under the `share/services` directory, and do not appear in the K menu.

A KParts component is described in a service desktop file, which specifies the filename of the library, the name of the component in it, the type of component, and the type of files handled by the component.

## Desktop file for the KParts component

The `.desktop` file describing the `KXMLTreePart` component can be named `kxmltreepart.desktop` (the name only matters for the `Makefile.am`) and reads as follows:

```
[Desktop Entry]
Type=Service
Name=XML Tree
MimeType=text/xml
ServiceTypes=KParts/ReadOnlyPart
X-KDE-Library=libkxmltreepart
```

The lines in the above `.desktop` file indicate the following:

- `Type=Service` defines a "service" (i.e., a component, not an application)
- `Name=XML Tree` assigns the service the name "XML Tree"
- `MimeType=text/xml` allows this component to open XML files for which the mimetype in KDE is `text/xml`
- `ServiceTypes=Kparts/ReadOnlyPart` means that the type of service provided by the component is "read-only part"
- `X-KDE-Library=libkxmltreepart` makes the library implementing this component/service `libkxmltreepart`

You can now run `make install` and the last line of `Makefile.am` installs the `.desktop` file into the appropriate location, `$prefix/share/services`.

## Selecting the preferred component for a type of file

There is already a component which is able to display XML files in Konqueror -- KHTML itself, the HTML engine that makes Konqueror such a good Web browser.

The first way of ensuring that `KXMLTreePart` is used is to right-click an XML file and to choose "XML Tree" under the "Preview In" submenu.

The second way is to make the XML tree the preferred component for XML files. For this, simply run `keditfiletype text/xml`, or use "Configure Konqueror..." / "File Associations" in Konqueror's menus, type `xml`, select the `text/xml` mimetype, switch

to the Embedding tab, and move up "XML Tree" in the list. After this, a simple left click on any `.xml` file in Konqueror opens it in the XML Tree component.

---

## Section 5. Adding actions to the component

### How to add functionality to a part

The `KXMLTreePart` component currently only features a widget. But KParts allows more than widget embedding; it also lets the part provide additional user interface items, such as menu items and toolbar buttons, which are merged into the hosting application's menus and toolbars. Those additional items allow the user to access additional functionality provided by the part that the widget can't provide directly.

The necessary steps to add functionality to the part are:

- Creating the actions, which give access to the functionality
- Implementing the functionality in methods (slots) called when the user activates the actions
- Defining the layout of the actions in the user interface using an XML file
- Declaring the XML file as the component's, which requires only a simple line in the constructor

### Creating actions

Creating an action means instantiating the class `KAction` or one of its derived classes. For instance, one instantiates a `KToggleAction` to implement an action that can be toggled on or off, such as a menu item with a checkmark.

`KSelectAction` lets the user select between several items; `KFontAction` lets the user select a font, etc.

The functionality to be added to the `KXMLTreePart` is a simple action that opens all the items in the tree. It takes one line to create the action (we've broken it into three lines here to display properly):

```
m_openCloseAll = new KAction( i18n("Open All"), 0, this,
    SLOT( slotOpenCloseAll() ), actionCollection(),
    "openCloseAll" );
```

The arguments are, in order: the text, the key shortcut, the object implementing the slot, the slot name, the parent, and the action name.

It is important to set the parent of the action to be the action collection provided by the part. This is the only way to make it find the action later on when building the GUI from the XML file.

The action will be turned into "Close All" after being used, hence the member variable. It is not needed to store pointers to all actions, only those which need to be changed later, e.g., changing the text, or enabling/disabling the action.

Add `class KAction;` and `KAction *m_openCloseAll` to the header file, to define the member variable.

## Implementing the Open/Close All action

The next step implements the slot that opens all items in the tree.

```
void KXMLTreePart::slotOpenCloseAll()
{
    bool open = m_openCloseAll->text() == i18n("Open All");
    QListViewItem* item = m_listView->firstChild();
    for ( ; item ; item = item->itemBelow() )
        item->setOpen( open );
    m_openCloseAll->setText( open ? i18n("Close All") : i18n("Open All") );
}
```

The last line changes the text of the action to "Close All" and back to "Open All" every time the action is used. The first line determines the action to take (opening or closing) from the text of the action.

Don't forget to declare the slot in the header file under a `protected slots:` heading for instance.

## Designing a user interface with XML

The XML file describes the layout of the menus and submenus in the menubar as well as the toolbars and the toolbar buttons. The menubar, menus, and toolbars are containers; menu items and toolbar buttons are the actions. Using XML GUI, the interface can be changed without recompilation of the program code, which is how the toolbar editor can alter the application's interface, for instance.

The XML file below, `kxmltreepart.rc`, specifies that the `KXMLTreePart` component's "openCloseAll" action should be inserted into the View menu.

```
<!DOCTYPE kpartgui>
<kpartgui name="KXMLTreePart">
  <MenuBar>
    <Menu name="view"><Text>&amp;View</Text>
      <Action name="openCloseAll"/>
    </Menu>
  </MenuBar>
</kpartgui>
```

## Breaking down the kxmltreepart.rc file

A closer look at the `kxmltreepart.rc` file shows that the `DOCTYPE` tag contains the name of the main element, which should be set to `kpartgui`. The top-level elements are `MenuBar` and `ToolBar`, the main containers of user interface items.

The menus are described in the `MenuBar`. They have names, which are used for merging. The standard way of naming menus is to use all lowercase: Konqueror defines "file", "edit", "view", "go", "tools", "settings", "window", and "help". Menus also have a text, which is displayed in the user interface, and can be translated. Toolbars work in a very similar manner, with a `ToolBar` tag for each one, a name and a text.

Actions are laid out inside the menus and toolbars. They are simply described by name. Those names are very important because they are used to match the actions created in the code.

## Installing the XML file

The last step is to give the framework the name of the XML file describing the GUI. The standard place for this file is under `$prefix/share/apps/instancename/`, and in this case, passing the filename without a path is enough. Simply add this line to the constructor:

```
setXMLFile( "kxmltreepart.rc" );
```

Konqueror in KDE 2.x only shows the part's actions if the part provides a `BrowserExtension` (for no particular reason, so this has been changed in KDE 3). Therefore the part needs to provide one, even a very simple one, with this line in the constructor:

```
// Browser integration (necessary for Konqueror-2.x to insert the actions)
(void) new KParts::BrowserExtension(this);
```

In order to install the XML file, the `Makefile.am` must read:

```
kxmltreedir = $(kde_datadir)/kxmltreepart
kxmltree_DATA = kxmltreepart.rc
```

The read-only part is ready. The next section explains how to extend a read-only part for more integration into Konqueror and then how to turn it into a read-write part.

---

## Section 6. Extending components

### The Browser Extension

If the `KXMLTreePart` component is meant to be used by Konqueror to display a certain type of file when clicking on it in the file manager, better integration with Konqueror can be provided by creating a `BrowserExtension` for the part.

This allows, for example, the component to save and restore its state when the user navigates using the back and forward actions in Konqueror, or for session management purposes. It also allows the component to request that a URL be opened by Konqueror. For example, this is used when clicking on a link in the HTML component.

To provide a browser extension for a better integration of a read-only part into Konqueror, the component usually needs to define a class that inherits from `KParts::BrowserExtension`. An instance of that class must be created by the part's constructor, with the part itself as the parent object. Konqueror will detect it and use it automatically.

### The BrowserExtension-derived class

The `BrowserExtension`-derived class should re-implement `xOffset()` and `yOffset()` if the component's widget has scrollbars, as in the case of it inheriting `QScrollView`. Reimplementing `saveState` and `restoreState` allows the component to store additional information in memory, such as when navigating in Konqueror using the back and forward actions.

The browser extension's signals (which are public, unlike usual signals) also allow a component, among other things, to open a URL (`openURLRequest`), to open a window (`createNewWindow`), to show a standard popup menu for a set of files (`popupMenu`), and to change the URL shown in the location bar (`setLocationBarURL`).

## Developing a read-write part

A read-write part must inherit `KParts::ReadWritePart` and implement `saveFile()`. It must also implement the read-only status in `setReadWrite(bool rw)`, usually disabling some actions when `rw` is false and enabling them when `rw` is true. Any action that modifies the data must be prevented when the part is in read-only mode.

When the user modifies the data in the part, it must call `setModified()`. This usually updates the caption bar to show `modified`, and enables the confirmation when closing without saving.

Make sure to specify that the part is read-write in the `.desktop` file, using `ServiceTypes=KParts/ReadWritePart`, and adapt the factory if using KDE 2 code (see [Factory code to create the part](#)) so that `setReadWrite` is called right after creating the component.

This is how to provide read-write functionality in a KParts component. Note that Konqueror can't use the read-write functionality: it is a design decision that Konqueror only uses read-only components. A separate application is needed to take advantage of the read-write functionality of the component.

# Resources

## Learn

- Read David's follow-on tutorial "[Creating KParts components, Part 2: Use KParts components in a KDE application](#)" (developerWorks, June 2002).
- To learn more about KParts, read David's article "[Coding with KParts](#)" (developerWorks, February 2002).
- For a thorough overview of KDE, read "[Introduction to KDE](#)" (developerWorks, May 2001).
- To understand XML when seen as a DOM tree, read "[Understanding DOM](#)" (developerWorks, July 2003).
- Read more about [Konqueror](#), KDE's file manager, Web browser, and universal viewer.
- Learn everything about KDE Development at the [KDE Developer's Corner](#).
- For a tutorial covering a read-write part, read the Chapter 12 of *KDE 2.0 Development: [Creating and Using Components \(KParts\)](#)*.
- Refer to the [API documentation for KParts::ReadOnlyPart](#) to follow the tutorial more easily.
- For detailed information on the signals and slots mechanism in the Qt toolkit, see [signals and slots](#).
- Find more [tutorials for Linux developers](#) in the [developerWorks Linux zone](#).
- Stay current with [developerWorks technical events and Webcasts](#).

## Get products and technologies

- Download [KDevelop](#), an integrated development environment (IDE) for KDE.
- Download [IBM trial software](#) directly from developerWorks.

## Discuss

- Read [developerWorks blogs](#), and get involved in the developerWorks community.

## About the author

### David Faure

David Faure is a French KDE Developer working for MandrakeSoft and the maintainer of Konqueror, the file manager and Web browser. He

also works on the KDE libraries (component technology and network transparency) and on KOffice (framework and KWord). His experience with KParts stems directly from his involvement in the design and development of KParts, particularly before the KDE 2.0 release. You can contact him at [faure@kde.org](mailto:faure@kde.org).