cube
scalasca

CubeGUI 4.7 | Plugin Developer Gu

How to develop a Cube GUI Plugin, road map and examples

August 2022
The Scalasca Development Team
scalasca@fz-juelich.de

JÜLICH
FORSCHUNGSZENTRUM

Attention

The Cube GUI User Guide is currently being rewritten and still incomplete. However, it should already contain enough information to get you started and avoid the most common pitfalls.

Contents

1 Copyright

2 Abstract

The version 4 of CUBE implementation provides support of varios types of Plugins.

This Guide helps an user to develop an additional standalone plugin which can be used within CUBE.

3 Cube Plugin Types

CUBE supports two types of plugins

- Context sensitive general Plugin
- Context free plugin

3.1 Context sensitive Cube Plugins

Plugins that derive from CubePlugin depend on a loaded cube file. They can react on user actions, e.g. tree item selection, and may insert a context menu or add a new tab next to the tree views. Examples for this type of plugins are the System Topology Plugin or the Statistics Plugin which are part of the Cube installation.

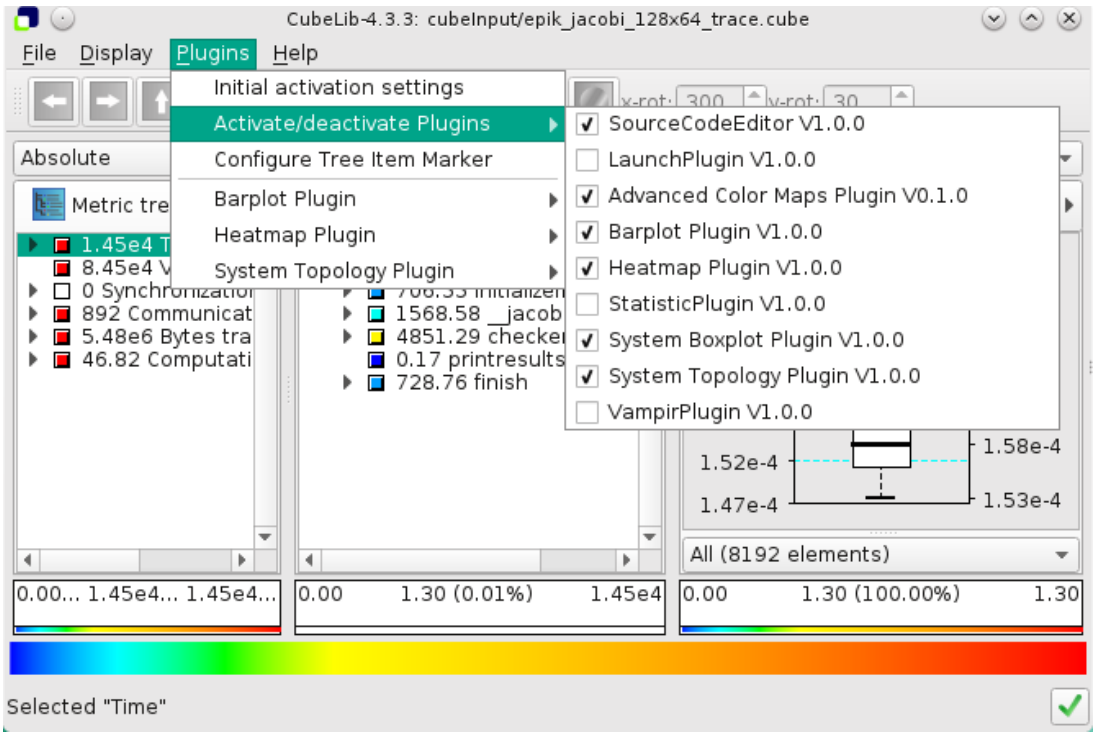

Figure 3.1: plugin menu

3.2 Context Free Plugins

Plugins that derive from ContextFreePlugin are only active if no cube file is loaded. These plugins create or modify Cube objects, which can be loaded and displayed.

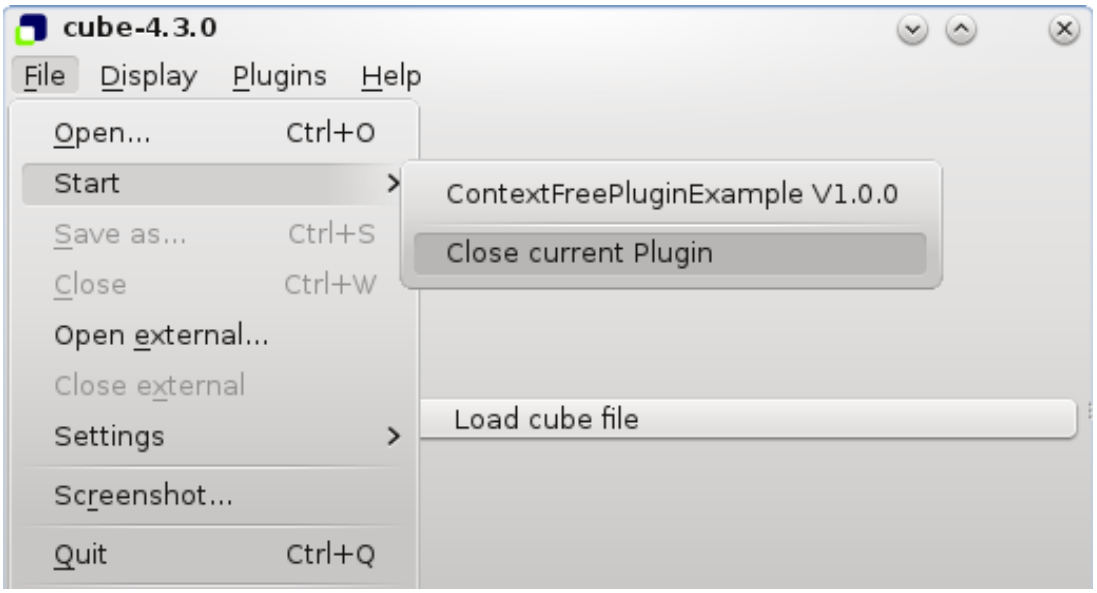

Figure 3.2: context free plugin menu

4 Step by step example for CubePlugin

The following sections describe the steps that are required to create a plugin. For simplicity, a separate project is created and the generated binary will to be copied to the plugin directory of the given cube installation.

Files of the simple cube plugin project:

- example-simple.pro.in
- SimpleExample.h
- SimpleExample.cpp

4.1 Qt project file

To create a cube plugin, a makefile and source files have to be generated. The makefile can be generated automatically from a Qt project file

First we specify the path to the "cube-config" script of the cube installation. This script delivers correct flags for compiling and linking.

```
CUBELIB_CONFIG = @CUBE_CUBELIB@

INCLUDEPATH    += $$system($$CUBEGUI_CONFIG --include)  $$system($$CUBELIB_CONFIG --include)
LIBS           += $$system($$CUBEGUI_CONFIG --ldflags) $$system($$CUBEGUI_CONFIG --libs)



TEMPLATE        = lib
CONFIG         += plugin
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

HEADERS         = SimpleExample.h
SOURCES         = SimpleExample.cpp
TARGET          = $$qtLibraryTarget(SimpleExamplePlugin)
```

qmake && make will build the first plugin example libExamplePluginSimple.so. The plugin will be copied to the plugin directory, e.g. /opt/cube/lib64/plugins.

4.2 SimpleExample.h

The example describes a minimal cube plugin, which is inserted as an additional tab next to the SystemTree. It shows the text of the recently selected tree item. The complete source of the example can be found in `$CUBE_INSTALL_PREFIX/share/doc/cube/example/gui/plugin-example`.

Every cube plugin has to derive from cubepluginapi::CubePlugin. To use Qt's signal and slot mechanism it also has to derive from QObject. If the plugin should be added as a tab next to a tree widget, it has to derive from cubegui::TabInterface.

```
class SimpleExample : public QObject, public cubepluginapi::CubePlugin, cubepluginapi::TabInterface
```

The class header is followed by the following macro definitions:

- `Q_OBJECT` is required to handle signals and slots.
- `Q_INTERFACES( cubepluginapi::CubePlugin )` tells Qt that the class implements the CubePlugin interface and generates the method `qt_metacast(char*)` to cast the plugin object to CubePlugin using the class name given as as character array.
- For Qt versions >= 5.0 the plugin has to be exported using the `Q_PLUGIN_METADATA()` macro. The unique plugin name "SimpleExamplePlugin" is assigned. For Qt versions < 5.0, `Q_EXPORT_PLUGIN2` has be be used (see Section 4.3).

```
class SimpleExample : public QObject, public cubepluginapi::CubePlugin, cubepluginapi::TabInterface
{
    Q_OBJECT
    Q_INTERFACES( cubepluginapi::CubePlugin )

#if QT_VERSION >= 0x050000
    Q_PLUGIN_METADATA( IID "SimpleExamplePlugin" ) // unique plugin name
#endif
```

The class SimpleExample has to implement the pure virtual methods from cubepluginapi::CubePlugin and cubegui::TabInterface.

```
public:
    SimpleExample();

    // CubePlugin implementation
    virtual bool
    cubeOpened( cubepluginapi::PluginServices* service );
    virtual void
    cubeClosed();
    virtual QString
    name() const;
    virtual void
    version( int& major,
             int& minor,
             int& bugfix ) const;
    virtual QString
    getHelpText() const;

    // TabInterface implementation
    virtual QString
    label() const;
    virtual QWidget*
    widget();
```

4.3 SimpleExample.cpp

SimpleExample.cpp

For Qt versions < 5.0, `Q_EXPORT_PLUGIN2` is used to export the plugin. The first argument is a unique name for the plugin, the second the name of the class.

```
using namespace cubepluginapi;
using namespace simpleexampleplugin;

#if QT_VERSION < 0x050000
Q_EXPORT_PLUGIN2( SimpleExamplePlugin, SimpleExample ); // ( PluginName, ClassName )
#endif
```

The function `cubeOpened(PluginServices* service)` is the starting point of our plugin. Allocation of data and GUI objects should be done here, not in the constructor. This allows to free the resources, if the plugin is deactivated.

Here we create the main widget, which should be added as a system tab. Our plugin derives from TabInterface, so `service->addTab(SYSTEMTAB, this)` can be called.

If the user selects a tree item, service will emit a corresponding signal. To react on this event, the signal has to be connected to the slot `treeItemIsSelected()` of our plugin class.

The function returns true, if the plugin should be started. It it returns false, the plugin is closed and deleted.

The function `cubeClosed()` is called if the cube file is closed or if the plugin is unloaded by the user. All resources which have been allocated in `cubeOpened` have to be deleted here.

```
bool
SimpleExample::cubeOpened( PluginServices* service )
{
    this->service = service;

    widget_ = new QWidget();
    qlabel_ = new QLabel( "example string" );
    QVBoxLayout* layout = new QVBoxLayout();
    widget_->setLayout( layout );
    layout->addWidget( qlabel_ );

    service->addTab( SYSTEM, this );

    connect( service, SIGNAL( treeItemIsSelected( cubepluginapi::TreeItem* ) ),
             this, SLOT( treeItemIsSelected( cubepluginapi::TreeItem* ) ) );
    return true; // initialisation is ok => plugin should be shown
}

void
SimpleExample::cubeClosed()
{
    delete widget_;
}
```

Each plugin has to set a version number. If several plugins with the same identifier (see function `name()`) exist, the one with the highest version number will be loaded.

```
void
SimpleExample::version( int& major, int& minor, int& bugfix ) const
{
    major  = 1;
    minor  = 0;
    bugfix = 0;
}
```

This function returns the unique plugin name. Only one plugin with this name will be loaded.

```
QString
SimpleExample::name() const
{
    return "Simple Example";
}
```

The following function returns a text to describe the plugin. It will be used by help menu of the cube GUI.

```
QString
SimpleExample::getHelpText() const
{
    return "Just a simple example.";
}
```

The following two functions contain the implementation of TabInterface.

The function `widget()` returns the QWidget that will be placed into the tab, which has been created with `service->addTab` in `initialize()`.

```
QWidget*
SimpleExample::widget()
{
    return widget_;
}
```

The function `label()` returns the label of the new tab.

```
QString
SimpleExample::label() const
{
    return "Example Plugin Label";
}
```

This method is a slot, which is called if a tree item is selected. The argument provides information about the selected item. With item->getDisplayType(), the location (METRIC, CALL, SYSTEM) can be identified.

```
void
SimpleExample::treeItemIsSelected( TreeItem* item )
{
    QString txt = item->getName() + " " + QString::number( item->getValue() );
    qlabel_->setText( txt );
}
```

5 Step by step example for ContextFreePlugin

The following sections describe the steps that are required to create a plugin which derives from ContextFreePlugin. For simplicity, a separate project is created and the generated binary will to be copied to the plugin directory of the given cube installation.

5.1 Qt project file

To create a cube plugins, a makefile and source files have to be generated. The makefile can be generated automatically from a Qt project file

First we specify the path to the "cube-config" script of the cube installation. This script delivers correct flags for compiling and linking.

```
CUBEGUI_CONFIG = @prefix@/bin/cubegui-config
CUBELIB_CONFIG = @CUBE_CUBELIB@

INCLUDEPATH    += $$system($$CUBEGUI_CONFIG --include) $$system($$CUBELIB_CONFIG --include)
LIBS           += $$system($$CUBEGUI_CONFIG --ldflags) $$system($$CUBEGUI_CONFIG --libs)


TEMPLATE        = lib
CONFIG         += plugin
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

HEADERS         = ContextFreePluginExample.h
SOURCES         = ContextFreePluginExample.cpp
TARGET          = $$qtLibraryTarget(ContextFreeExamplePlugin)
```

qmake && make will build the first plugin example libContextFreeExamplePlugin.so. The plugin will be copied to the plugin directory, e.g. /opt/cube/lib64/plugins.

5.2 ContextFreePluginExample.h

The example ContextFreePluginExample.h describes a minimal context free plugin. The plugin becames active, if Cube is started without an input file, or if the cube file is closed.

The complete source of the example can be found in `$CUBE_INSTALL_PREFIX/share/doc/cube/example/gui/cont`

A context free plugin has to derive from ContextFreePlugin. To use Qt's signal and slot mechanism it also has to derive from QObject.

```
class ContextFreePluginExample : public QObject, public cubepluginapi::ContextFreePlugin
```

The class header is followed by the following macro definitions:

- `Q_OBJECT` is required to handle signals and slots.
- `Q_INTERFACES( ContextFreePlugin )` tells Qt that the class implements the ContextFreePlugin interface and generates the method `qt_metacast(char*)` to cast the plugin object to ContextFreePlugin using the class name given as as character array.
- For Qt versions >= 5.0 the plugin has to be exported using the `Q_PLUGIN_METADATA()` macro. The unique plugin name "ContextFreePlugin" is assigned. For Qt versions < 5.0, `Q_EXPORT_PLUGIN2` has be be used (see Section 5.3).

```
class ContextFreePluginExample : public QObject, public cubepluginapi::ContextFreePlugin
{
    Q_OBJECT
    Q_INTERFACES( cubepluginapi::ContextFreePlugin )
#if QT_VERSION >= 0x050000
    Q_PLUGIN_METADATA( IID "ContextFreePluginExample" )
#endif
```

The class ContextFreePluginExample has to implement all pure virtual methods from ContextFreePlugin.

```
public:
    // ContextFreePlugin interface
    virtual QString
    name() const;

    virtual void
    opened( cubepluginapi::ContextFreeServices* service );

    virtual void
    closed();

    virtual void
    version( int& major,
             int& minor,
             int& bugfix ) const;

    virtual QString
    getHelpText() const;

private slots:
```

5.3 ContextFreeExample.cpp

ContextFreePluginExample.cpp
For Qt versions < 5.0, `Q_EXPORT_PLUGIN2` is used to export the plugin. The first argument is a unique name for the plugin, the second the name of the plugin class.

```
using namespace contextfreepluginexample;
using namespace cubepluginapi;

#if QT_VERSION < 0x050000
Q_EXPORT_PLUGIN2( ContextFreePluginExample, ContextFreePluginExample ) // ( PluginName, ClassName )
#endif
```

The function `opened(ContextFreeServices* service)` is the starting point of our plugin. With `service->getWidget()` we get a widget on Cube's main screen, in which we can

place the GUI elements of our plugin. In this example, only one button will be placed on the main screen. Activation of this button will call the slot function `startAction()`.

```
void
ContextFreePluginExample::opened( ContextFreeServices* service )
{
    this->service = service;
    qDebug() << "context free plugin opened";

    QWidget*     widget = service->getWidget();
    QVBoxLayout* layout = new QVBoxLayout();
    widget->setLayout( layout );

    QPushButton* but = new QPushButton( "Load cube file" );
    layout->addWidget( but );

    connect( but, SIGNAL( clicked() ), this, SLOT( startAction() ) );
}
```

The function `closed()` is called if the plugin gets inactive because a cube file is loaded or the Cube GUI is closed. All resources which have been allocated in `opened()` have to be deleted here.

```
void
ContextFreePluginExample::closed()
{
    qDebug() << "context free plugin closed";
    // free all resources allocated in ContextFreePluginExample::opened()
    // children of service->getWidget() will be deleted automatically
}
```

This function is called, if the user clicks on the Button "Load cube file". Usually, a context free plugin will create cube data. In this small example, it simply loads the cube file which is choosen from a file dialog.

```
void
ContextFreePluginExample::startAction()
{
    QString openFileName = QFileDialog::getOpenFileName( service->getWidget(),
                                                         tr( "Choose a file to open" ),
                                                         "",
                                                         tr( "Cube3/4 files (*cube *cube.gz *.cubex);;Cube4
       files (*.cubex);;Cube3 files (*.cube.gz *.cube);;All files (*.*);;All files (*)" ) );
    std::vector<std::string> fileNames;
    fileNames.push_back( openFileName.toStdString() );
    cube::CubeProxy* cube = cube::CubeProxy::create( cube::ALGORITHM_EMPTY, fileNames );
    service->openCube( cube ); // will be deleted automatically, if user closes cube
}
```

Each plugin has to set a version number. If several plugins with the same identifier (see function `name()`) exist, the one with the highest version number will be loaded.

```
void
ContextFreePluginExample::version( int& major, int& minor, int& bugfix ) const
{
    major  = 1;
    minor  = 0;
    bugfix = 0;
}
```

This function returns the unique plugin name. Only one plugin with this name will be loaded.

```
QString
ContextFreePluginExample::name() const
{
    return "Context Free Demo";
}
```

The following function returns a text to describe the plugin. It will be used by help menu of the cube GUI.

```
QString
ContextFreePluginExample::getHelpText() const
{
    return "context free plugin help text";
}
```

6 Developing plugins

6.1 Further examples

6.1.1 Extensive example

The example in `$CUBE_INSTALL_PREFIX/share/doc/cube/example/gui/plugin` uses all major functions of PluginServices. It contains functions to handle

- settings, global preferences e.g. number formats
- further tab functions
- selections
- menus, context menus and toolbars
- global values to communicate with other plugins

See also
cubepluginapi::PluginServices
DemoPlugin.h
DemoPlugin.cpp
demo-plugin.pro.in

6.1.2 ValueView example

`$CUBE_INSTALL_PREFIX/share/doc/cube/example/gui/value-view` is an example of a plugin that offers an additional value view. It adds a boxplot view for tau metrics. The example

- implements cubepluginapi::ValueView and replaces the colored box with a boxplot
- adds a user dialog which allows the user to change the icon size
- implements cubepluginapi::SettingsHandler to save the icon size settings

See also
cubepluginapi::PluginServices
TauValueView.h
TauValueView.cpp
tau-value.pro.in

6.2 Problems loading plugins

If the plugin doesn't load, start cube with `-verbose` to get further information. The most likely reason is an undefined reference:

```
plugin /opt/cube/lib64/cube-plugins/libSimpleExamplePlugin.so is not a valid
    CubePlugin version cubeplugin/1.1
Cannot load library /opt/cube/lib64/cube-plugins/libSimpleExamplePlugin.so:
    (undefined symbol: _ZN13SimpleExample10cubeClosedEv)
```

If we remove the definition of the method `cubeClosed()` in SimpleExample.cpp, the plugin is created without errors, but it cannot be loaded. `cube -verbose` shows the error message above.

When building plugins, it is important to ensure that the plugin is configured in the same way as cube. A plugin build with incompatible options shows the following error:

```
Plugin verification data mismatch in '/opt/cube/lib64/cube-plugins/libSimpleExamplePlugin.so'
```

The same compiler, the same Qt library and the same configuration options have to be used. Only plugins which are created using a Qt library with a lower minor version can also be loaded.

7 Usage of the Cube Plugin API

The class cubepluginapi::PluginServices is used by the plugins to interact with the cube GUI. This Chapter will provide an overview about the most important functions.

7.1 Functions to show information provided by the plugin

7.1.1 Add a new Tab next to the System tree

To add one ore more tabs next to the system tree, the plugin has to call cubepluginapi::PluginServices::addTab. This function requires a cubegui::TabInterface as parameter. The tab has to define a label and a widget. See 4 for a simple demo.

7.1.2 Add a context menu to a tree view

A context menu is shown, if the user clicks with the right mouse button on a tree. With cubepluginapi::PluginServices::addContextMenuItem, the plugin can add a menu item to the context menu.

7.1.3 Create a toolbar

Cube provides toolbars, if they are enabled (e.g. Preferences, Synchronisation). A plugin may create additional toolbars with cubepluginapi::PluginServices::addToolBar. A toolbar may be assigned to a tab. In this case, the toolbar is only visible, if the tab is also visible. If the tab gets detached, the toolbar will be moved to the new window.

7.1.4 Define a shortcut

Shortcuts can be defined with QAction::setShortcut. To ensure that the plugin shortcuts don't interfere with the default shortcuts, the context should be set using QAction::setShortcutContext(Qt::WidgetWithChildrenShortcut).

7.1.5 Create an additional colormap

The colormap plugin example (ColorMapPlugin.h, ColorMapPlugin.cpp) demonstrates how to use cubepluginapi::PluginServices::addColorMap

7.1.6 Create an alternative value view

The default value view shows a colored square next to the numerical value. The function cubepluginapi::PluginServices::addValueView adds the given value view to the list of available views from which the user can choose the active one. See 6.1.2 for an example implementation.

7.1.7 Add a metric to the metric tree

To add a new metric to the metric tree, call cubepluginapi::PluginServices::addMetric.

7.1.8 Add a marker to a tree

A plugin may define one or more tree item marker to tag items of interest.

Tree items are marked in different ways:

- Items with a colored background show that a plugin has set a marker
- Items with a colored frame indicate that a collapsed child has been marked.
- Items with a black frame indicate that there are several collapsed children with different marker.
- Items with a dotted frame show a dependency. A marked item of the right neighbor tree depends on
- Items can be grayed out. These items are either marked as unimportant by a plugin, or the user has choosen to gray out all items, for which no marker is set. this item. The dependent item is only marked, if the dotted item is selected.

To create a new marker, cubepluginapi::PluginServices::getTreeItemMarker has to be called. Then, the marker cann be added in two different ways:

- for one TreeItem
  cubepluginapi::PluginServices::addMarker(TreeItem ∗item, const TreeItemMarker ∗marker, bool isDependency)
- for a set of dependent items
  cubepluginapi::PluginServices::addMarker(const TreeItemMarker ∗marker, TreeItem ∗metric, TreeItem ∗call, TreeItem ∗system)

See 6.1.1 for an example implementation.

7.1.9 Add a status message

To write a message to the status line at the bottom of the cube window, call cubepluginapi::PluginServices::setMessage.

7.1.10 Communication with other plugins

To communicate with other plugins, a named value can be send with cubepluginapi::PluginServices::setGlobalValue. After a value has been set, the SIGNAL cube-

pluginapi::PluginServices::globalValueChanged is emitted and the value can be read by other plugins. After a plugin is successfully started, the global value <plugin name>=""><::started is set to true.

7.2 Parallel execution of compute-intensive tasks

The API function cubepluginapi::PluginServices::createFuture(TabInterface* tab = 0) creates a Future object to execute a task in parallel. The Future object is deleted after the plugin has been closed. If a TabInterface is given, a progress bar will be displayed in the given tab while the tasks are running.

The parallel plugin example (ParallelPlugin.h, ParallelPlugin.cpp) demonstrates how to use cubepluginapi::Future

www.scalasca.org