GOI SAMPLE/Skeleton PLUGIN

Introduction

Sample plugin is based on the GTKSocket and GTKPlug approach for GOI supported programming languages such as Python, Javascript.

The GTKSocket part has been developed in the C programming language due to the absence of registration methods through python or javascript.

The GTKPlug part can be developed in any of the GOI supported programming languages. It is a place where you will have to develop the main logic of the program except XFCE signal handling.

Useful information related to different Files

Panel-plugin Plug Programming language Selection

Now, You have to select one programming language either python or javascript, to move forward with the GTKPlug approach.

If you are choosing out of python, then move forward with **panel-plugin/python/plugin.py** deleting out

panel-plugin/javascript/plugin.js, as it would have no use inside the python based plugin.

panel-plugin/python/<your-selected-programming-language-file>

This is the main file where all logic related to GTKPlug will be implemented.For example, “Sample Plugin” label have been added inside the GTKPlug through “SamplePlugin” class.

Class SamplePlugin is a main class for sample GTKPlug, it contains logic for GTKPlug and its signals.

GTKSocket id have been received through **sys.argv[1] (in python) and Number(ARGV) (in javascript).**

COMMON FILES FOR SAMPLE PLUGIN

Waf

‘waf’ is a generated file so you don’t have to do much with this generated file. All installation operations will be performed through the ‘wscript’ file.

Wscript

This file is an actual file that is used for actual installation, which is based on 6 point installation process those are given below.

- **configure:** it is used to configure the project and find the location of the prerequisites, for example if you want to load a C compiler you can define over this step.
- **build:** it is used to transform the source files into build files
- **install:** used to install the build files
- **uninstall:** used for un-installation of the build files
- **dist:** create an archive of the source files

- **clean:** remove the build files.

```
APPNAME = 'Enter your plugin name'
VERSION = 'Enter your plugin version'

top = '.'
out = 'build'

def options (ctx):
    ctx.load('compiler_c')

def configure (ctx):
    args = '--cflags - libs'
    ctx.check_cfg(package = 'gtk+-3.0', atleast_version = '3.22',
        uselib_store = 'GTK', mandatory = True, args = args)


def build (ctx):
    ctx.program(
        features     = 'c cshlib',
        is_lib       = True,
        source       = ctx.path.ant_glob('panel-plugin/*.c'),
        packages     = 'gtk+-3.0',
        target       = 'sample-plugin',
        install_path = '${LIBDIR}/xfce4/panel/plugins/',
        uselib       = 'GTK',
        )

    ctx(
        features = 'subst',
        source = 'panel-plugin/sample.desktop.in',
        target = 'panel-plugin/sample.desktop')

    ctx.install_files(
        'usr/share/xfce4/panel/plugins/',
        'panel-plugin/sample.desktop')

def checkinstall (ctx):
    ctx.exec_command('checkinstall - pkgname=' + APPNAME +
     ' - pkgversion=' + VERSION + ' - provides=' + APPNAME +
     ' - deldoc=yes - deldesc=yes - delspec=yes - backup=no' +
     ' - exclude=/home -y ./waf install')l')
```

def options(ctx) is used to load the C compiler. If you are using some compiled language in your plugin, you can load the compiler through this function.
You can take arguments(can set flags) during installation through this function.

def configure(ctx) is used to check any specific version of the library.

```
ctx.check_cfg(package = 'libxfce4panel-2.0', atleast_version = '4.12',
            uselib_store = 'XFCE4PANEL', mandatory = True, args = args)
```

From the above code snippet, you can see various arguments as conditions for validation of a specific library. For example with mandatory arguments you can set whether the installation of a library is mandatory or not.

def build(ctx) is the main function to install different files and setup plugin configuration. Program takes several arguments through which you can set up configuration. For example through uselib you can configure libraries on which the plugin is dependent. install_files() takes two mandatory arguments as string, Through first argument system path of file that have to install is pass off, and then through second argument actual path of file is pass off.

def checkinstall(ctx) you can verify the installation of the plugin.

Note: If you are getting confused with ctx don't worry about it just a variable name you can replace ctx with your custom variable name also.

panel-plugin/sample.desktop.in

```
[Xfce Panel]
Type=X-XFCE-PanelPlugin
Encoding=UTF-8
Name=Sample Plugin
Comment=Sample plugin for the Xfce panel
```

```
Icon=xfce4-sample-plugin
X-XFCE-Module=sample-plugin
X-XFCE-Internal=false
X-XFCE-Unique=false
X-XFCE-API=2.0
```

If the module should have no more than 1 instance running at the same

time, you add this line:

```
X-XFCE-Unique=true
```

If the plugin is compatible with GTK+ 3, you need to add this line:

```
X-XFCE-Unique=true
```

Icon is used to define the application icon

```
Icon=gtk-icon-name
```

panel-plugin/sample.c

This file has been used to create GTKSocket for the XFCE-panel plugin. You can consider this file as a main file for GTKSocket related operations.

Plugin registration

To register a plugin with the plugin system there is one macro available that should be used, instead of using the library functions directly.

```
/* register the plugin */
XFCE_PANEL_PLUGIN_REGISTER(sample_construct);
```

The 'sample_construct' argument is the name of a function that may be cast to XfcePanelPluginFunc, i.e. it takes a single XfcePanelPlugin pointer as argument. In the function GTKSocket widget should be created and callbacks connected to the appropriate plugin signals.

Till now, registration of panel plugin is not possible with all GOI supported language, as you won't be able to access XFCE_PANEL_PLUGIN_REGISTER through python or Javascript, that why, it became little complex process with GTKSocket and GTKPlug approach with two programming language i,e GTKSocket in C and GTKPlug in Python/Javascript.

```
sprintf(BUFFER, "/usr/share/xfce4/panel/plugins/plugin %d &",
gtk_socket_get_id(sample->socket));
```

This line creates the whole system command as a single string and let us help to embed our GTKPlug into the GTKSocket. Then it is executed with the help of the system.

```
system(BUFFER)
```

panel-plugin/sample.h

sample.h is a header file which is used for function declarations and macro definitions that have to be shared between several source file.

```
/* plugin structure */
```

```
typedef struct
{
    XfcePanelPlugin *plugin;

    /* panel widgets */
    GtkWidget *socket;

} SamplePlugin;
```

As you can see in the above code snippet, we have defined two things, i.e XfcePanelPlugin pointer and GTKSocket pointer under single struct SamplePlugin.

So, If you want to access these pointers, you can access them in this way, **SamplePlugin->socket** or **SamplePlugin->plugin**.

File Based on Plug Programming Language

panel-plugin/python/plugin.py

For the Python Programming language, we will be moving forward with this file only. There is no need to import panel-plugin/javascript/plugin.js, you can ignore that file.

```
socket_id = int(sys.argv[1])
```

Through sys.argv, we have received an socket id. With the help of socket_id, we will create Gtkplug.

```
class SamplePlugin:

    def __init__(self, socket_id):
        # create new plug with socket_id
        self.plug = Gtk.Plug.new(socket_id)

        # create sample plugin label
        self.label = Gtk.Label(label = "Sample Plugin")
```

```
        self.plug.add(self.label)

        # added Gtkplug signals
        self.plug.connect("embedded", self.embed_event)
        self.plug.connect("destroy", Gtk.main_quit)

        # show plug
        self.plug.show_all()


    def embed_event(self, widget):
        # event function that will be called after embedded signal
        print("A plug has been embedded")
```

In the SamplePlugin class, we have created a plug with the help of GTKSocket id, and then the “Sample Plugin” Label is added to the gtk plug. Along with it two gtkplug signals are also added i.e “embedded” and “destroy”.

When a plug is successfully embedded into the GTKSocket, “A plug has been embedded” will get printed on the debugger.

panel-plugin/javascript/plugin.js

If you are creating a sample plugin with javascript, then this file will be useful for you. **panel-plugin/python/plugin.py** can be ignored in this case will not get install at the time of installation.

```
app.application.run (ARGV);
```

Through ARGV, arguments have been passed through the application.

```
class SamplePlugin {

    /* Create the application itself
       This boilerplate code is needed to build any GTK+ application. */
    constructor() {
        this.application = new Gtk.Application ({
            application_id: 'org.example.sampleplugin',
```

```
            flags: Gio.ApplicationFlags.FLAGS_NONE
        });

        // Connect 'activate' and 'startup' signals to the callback functions
        this.application.connect('activate', this._onActivate.bind(this));
        this.application.connect('startup', this._onStartup.bind(this));
    }

    // Callback function for 'activate' signal presents windows when active
    _onActivate() {
        Gtk.main();
    }

    // Callback function for 'startup' signal initializes menus and builds the
UI
    _onStartup() {
        this._buildUI();
    }

    // Build the application's UI
    _buildUI() {

        // Create the application plug
        this.plug = Gtk.Plug.new(Number(ARGV))

        // Create the label
        this.label = new Gtk.Label({ label: "Sample Plugin" });

        // Add label to Plug
        this.plug.add(this.label);

        // Show the plug and all child widgets
        this.plug.show_all();

    }
};
```

On the initialization of Sample plugin class, we have two different signals i.e “activate” and “startup”

Activate: it will have a callback function, which gets called when the window gets active.

Startup: it is used to initialize menus and buildUI.

_buildUI() is associated with the callback function of the startup signal. Under _buildUI(), plug have been created with the usage of socket id. Then a “Sample Plugin” label has been added to the plug.

Steps to use sample plugin

Step 1: Download or git clone sample plugin onto the local computer.

Step 2: According to the selected programming language, You can add changes to the plug.

Step 3: Configure your plugin with a given command in the terminal.

```
./waf configure --prefix=/usr
./waf build --verbose
```

Step 4: You can install a specific programming language plug with a given command.

For installation of Javascript Plug use following command:

```
./waf install -p 'js'
```

For installation of Python Plug use the following command.

```
./waf install -p 'py'
```

Note: Default installation would be Python plug if you missed out '-p' flag

Now Sample Plugin has been successfully installed into the xfce panel, you can confirm it by viewing down xfce panel.