# Tutorial: Writing Plugins

## I. Introduction

Writing plugins is simply a matter of creating a *Class Library* that contains a definition of at least one class that implements one of the *IPlugin*-derived interfaces that are found in the *FractalBlaster.Universe* namespace.

## II. Preparation

- First create a project in Visual Studio 2010 of the type *Class Library*.
- Next you need to add a reference to the *FractalBlaster.Universe* assembly. This can either be added as a project (if your developing in the main solution) or you can browse for the assembly named FractalBlaster.Universe.dll.
- If you are developing in the main solution you will need to change your build location in order for the main application to load your plugin. To do this in a C# project, right-click on the project and open up the "Properties." Choose the "Build" tab on the left and near the bottom of this page, change the value of "Output path" to:

**..\build\debug\bin\Plugins**

## III. Creating the Plugin

The initial steps toward creating a plugin are the same regardless of the type of plugin being created. This section will explain the steps that must be performed for all plugins.

1. Create a class in the project you created. This class will be the class implementing one of the plugin interfaces in the next section; however, we are going to ignore that aspect in order to concentrate on the generic aspects.

2. Add the *PluginAttribute* to your class, passing at the bare minium the *Name* and *Description* of your plugin as arguments. The *Name* and *Description* is the recommended bare minimum, but you're encouraged to add as many of the *PluginAttribute* properties as you are able to. Also, I have yet to find a simple way to make an *Attribute* required by an interface, so the incusion of the *PluginAttribute* is not enforced; however, it should still be considered a requirement.

3. Implement the *IPlugin* interface. The *IPlugin* interface defines a single member, namely the *void Initialize(AppContext context)* method. This method is called on every plugin loaded to give the plugin some information about the environment it's running in. This information is passed to the plugin via the *AppContext* class. I will explain more on this later, all you need to know for now is that there's a good chance you will want to reference the context passed in at a later point in time, so it is a good idea to store it. In your class, create a field or property of type *AppContext* and in the *Initialize()* method, set it to the value of the context passed in. This completes an initial implementation of the *IPlugin* interface.

Example:

```
using FractalBlaster.Universe;

[PluginAttribute(Name="My Plugin", Description="Does stuff")]
public class MyPlugin {

    public void Initialize(AppContext context) {
        Context = context;
    }

    private AppContext Context { get; set; }
}
```

Advanced: The *AppContext* class has members allowing the plugin to get information regarding the other plugins running and most importantly, access to the *Engine*. The most common action a plugin might want to perform is finding out when a song has changed, a playlist was loaded, etc. The *Engine* property of the *AppContext* gives all plugins access to the currently running *Engine*, which allows the plugin to register with any events they are interested in.

# IV. Writing the Plugin

The next step is to decide which plugin you are going to write and implement the members defined in that plugin's interface. This section will cover each of the available plugins. Once you decide which type of plugin you're going to write, make your class extend that interface (if you haven't already done so) and provide implementations to each of it's members.

## a) IViewPlugin

Provides a user interface element to the application.

- ***Form UserInterface { get; }***
  This property is the only member defined in the *IViewPlugin* interface and should return an instance of the *System.Windows.Forms.Form* class that represents your user interface. Every view plugin is automatically added to the Views menu in the application and will be opened in a seperate window when selected.

## b) IEffectPlugin

Provides an effect to the audio being played by modifying the audio stream after decoding it, but before outputting it.

- ***MemoryStream** ProcessStream(**MemoryStream** stream)*
  Method that does the modifying of the audio stream. It is passed a stream that should be processed in some way to provide an effect and returned when finished.

## c) IMetadataPlugin

Provides a implementation for reading file metadata and properties. All metadata plugins are ran behind the scenes when creating an instance of *MediaFile*.

- ***IEnumerable<String>** SupportedFileExtensions { get; }*
  Property that returns an enumeration of file extensions supported by the plugin.

- ***IEnumerable<MediaProperty>** Analyze(**MediaFile** file)*
  Method that takes in a *MediaFile,* analyzes it and returns an enumeration of *MediaProperty*'s representing the metadata that it read

## d) IPlaylistPlugin

Plugin that provides an implementation for reading and writing playlists in different formats.

- ***IEnumerable<String>*** *SupportedFileExtensions { get; }*
  Property that returns an enumeration of file extensions supported by the plugin.

- ***Playlist*** *Read(**String** path)*
  Method that reads a file at the specified path and returns an instance of *Playlist* representing the contents that were read.

- *void Write(**Playlist** playlist, **String** path)*
  Method that writes the specified playlist to the specified path .

## e) IInputPlugin

Method that provides the logic necessary to input a file, including decoding of the file if necessary.

- ***MediaFile*** *OpenMedia(**String** path)*
  Opens the file at the specified path, prepares to input it and returns the opened file as a *MediaFile.*

- *void CloseMedia()*
  Closes the media file previously opened by *OpeneMedia(),* if any.

- *void SeekBeginning()*
  Resets the input stream of the currently opened media file back to the beginning.

- ***MemoryStream*** *ReadFrames(**Int32** count)*
  Reads the specified number of frames from the currently opened media file and returns them as a stream.

## f) IOutputPlugin

Plugin that provides an implementation for the outputting of a media file.

- ***Boolean*** *IsPlaying { get; }*
  Property that returns whether or not the plugin is currenting outputting media.

- ***Boolean*** *IsPaused { get; }*
  Property that returns whether or not the plugin is currently pausing output.

- *void Play()*
  Method that plays the current media file loaded in the *Engine.*

- *void Stop()*
  Method to stop the output of the currently loaded media file in the *Engine.*

- *void Pause()*
  Method to pause the output of the currently loaded media file in the *Engine.*

- *void Resume()*
  Method to resume the output of a media file that was previously paused.

## V. Finishing up

At this point the plugin is complete. To test the plugin, make sure it has been added to the *Plugins* directory that contains the FractalBlaster.exe executable. This will be done automatically if you followed the steps in the *Preparation* section for changing your build path.

Example:

```
using FractalBlaster.Universe;
using System.Windows.Forms;

[PluginAttribute(Name="My Plugin", Description="Does stuff")]
public class MyPlugin : IViewPlugin {

    public Form UserInterface { get; private set; }

    public void Initialize(AppContext context) {
        Context = context;
        Context.Engine.OnMediaChanged +=
             new MediaChangeHandler(SetSongPlaying);
        Display = new Label();
        Display.Text = "Hello, world! ...no song yet!";
        UserInterface = new Form();
        UserInterface.Controls.Add(Display);
    }

    private void SetSongPlaying(MediaFile song) {
        Display.Text = song.Metadata["Title"].Value.ToString() ?? "";
    }

    private AppContext Context { get; set; }
    private Label Display { get; set; }
}
```