Plugin file format - Element reference

Table of contents

1. Element plugin

```
<plugin
    id        = identifer
    version   = version string
    lazy?     = boolean default "false">
    <!-- Content: ANY -->
</plugin>
```

The `plugin` element is the root of any plugin file.

1.1. Attribute id

This is the identifier of the plugin.

1.2. Attribute version

1.3. Attribute lazy

A lazy plugin jumps over the brown fox.

2. Element variable

```
<variable
    name      = string
    value     = string>
    <!-- Content: EMPTY -->
</variable>
<!-- Contained by: plugin -->
```

Variables contribute to the namespace of variable expansion for when the context is the current plugin.

The value can contain other variable expansions in the context of the plugin. However, unlike normal variable expansion, which can use the values of any variables declared in the plugin, this element can only use the values of variable previously declared. This is simply because when plugins are loaded, the variables are processed first, making them available to other elements no matter where in the plugin they are found. However, while the variables are being processed, each only has the values of previously processed variables to work with.

2.1. Attribute name

Name of the variable. May not contain the characters `'{'`, `'}'` or `'$'`.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

2.2. Attribute value

Value for the variable.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

3. Element library

```
<library
    name?     = string
    path      = string>
    <!-- Content: ( setup | start | run | stop | shutdown )* -->
</library>
<!-- Contained by: plugin -->
```

This element indicates that the plugin has a runtime shared library associated with it. When the plugin is determined to be required (either by not having its `lazy` attribute set to `true` or by some dependency from another plugin, the shared library will be loaded.

Functions in the shared library can be called during the different lifecycle phases of the plugin system. These functions are specified by the children of the `library` element. The order of calls to these functions is the same as the lexicographical ordering in the element (within each lifecycle phase). Similary, if the plugin declares multiple `library` elements, they are processed in their lexicographical order.

Other elements in the plugin file may make use of symbols found in the shared libraries of the plugin. Usually, they will provide the option of specifying a library `name` attribute. If no library name is given then they search through the list of libraries, in order, attempting to find an appropriately named symbol, using the first that they find. However, if a `name` attribute is given, they will search only in the `library` element with the matching name. This means that in the typical case of only having one shared library, no library names need to be specified, but that possibility exists for disambiguating in the case of multiple `library` elements.

3.1. Attribute name

Name of the library to be used by other elements which wish to find symbols in the library.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

3.2. Attribute path

Path to the shared library.

Typically, plugins will be organised as a directory with a `plugin.xml` file and some number of shared libraries (although this need not be the case). It is often convenient, then to specify the path to the shared library with variable expansion:

```
<library path="${plugin.dir}/library.so"/>
```

This attribute is variable expanded. Variables are resolved in the context of the plugin.

4. Element setup

```
<setup
    symbol?   = string default "Plugin_setup">
    <!-- Content: EMPTY -->
</setup>
<!-- Contained by: library -->
```

This element specifies a method to be run during the setup phase of the plugin system.

The lifecycle method is found by searching for the given symbol in the `library` that contains the element. The method must follow the prototype:

```
bool (*)( Plugin* plugin )
```

This function should return `TRUE` if the plugin is successfully initialised. Otherwise, it should return `FALSE` and the plugin system will fail to start.

The method will only be called if the plugin is deemed to be required (either not having its `lazy` attribute set to `TRUE` or by a dependency from another required plugin.

4.1. Attribute symbol

The attribute gives the name of a symbol in the shared library pointed to by the parent `library`'s `path` attribute.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

5. Element start

```
<start
    symbol?   = string default "Plugin_start">
    <!-- Content: EMPTY -->
</start>
<!-- Contained by: library -->
```

This element specifies a method to be run during the start phase of the plugin system.

The lifecycle method is found by searching for the given symbol in the `library` that contains the element. The method must follow the prototype:

```
bool (*)( Plugin* plugin )
```

This function should return `TRUE` if the plugin is successfully initialised. Otherwise, it should return `FALSE` and the plugin system will fail to start.

The method will only be called if the plugin is deemed to be required (either not having its `lazy` attribute set to `TRUE` or by a dependency from another required plugin.

5.1. Attribute symbol

The attribute gives the name of a symbol in the shared library pointed to by the parent `library`'s `path` attribute.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

6. Element run

```
<run
    symbol?   = string default "Plugin_run">
    <!-- Content: EMPTY -->
</run>
<!-- Contained by: library -->
```

This element specifies a method to be run during the run phase of the plugin system.

The lifecycle method is found by searching for the given symbol in the `library` that contains the element. The method must follow the prototype:

```
bool (*)( Plugin* plugin )
```

This function should return `TRUE` if the plugin is successfully initialised. Otherwise, it should return `FALSE` and the plugin system will fail to start.

The method will only be called if the plugin is deemed to be required (either not having its `lazy` attribute set to `TRUE` or by a dependency from another required plugin.

6.1. Attribute symbol

The attribute gives the name of a symbol in the shared library pointed to by the parent `library`'s `path` attribute.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

7. Element stop

```
<stop
    symbol?   = string default "Plugin_stop">
    <!-- Content: EMPTY -->
</stop>
<!-- Contained by: library -->
```

This element specifies a method to be run during the stop phase of the plugin system.

The lifecycle method is found by searching for the given symbol in the `library` that contains the element. The method must follow the prototype:

```
bool (*)( Plugin* plugin )
```

This function should return `TRUE` if the plugin is successfully initialised. Otherwise, it should return `FALSE` and the plugin system will fail to start.

The method will only be called if the plugin is deemed to be required (either not having its `lazy` attribute set to `TRUE` or by a dependency from another required plugin.

7.1. Attribute symbol

The attribute gives the name of a symbol in the shared library pointed to by the parent `library`'s `path` attribute.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

8. Element shutdown

```
<shutdown
    symbol?   = string default "Plugin_shutdown">
    <!-- Content: EMPTY -->
</shutdown>
<!-- Contained by: library -->
```

This element specifies a method to be run during the shutdown phase of the plugin system.

The lifecycle method is found by searching for the given symbol in the `library` that contains the element. The method must follow the prototype:

```
bool (*)( Plugin* plugin )
```

This function should return `TRUE` if the plugin is successfully initialised. Otherwise, it should return `FALSE` and the plugin system will fail to start.

The method will only be called if the plugin is deemed to be required (either not having its `lazy` attribute set to `TRUE` or by a dependency from another required plugin.

8.1. Attribute symbol

The attribute gives the name of a symbol in the shared library pointed to by the parent `library`'s `path` attribute.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

9. Element requires

```
<requires
    plugin?   = plugin id
    point?    = extension point id
    version?  = version string
    match?    = { "perfect" | "equivalent" | "compatible" |
"greaterOrEqual" } default "compatible">
    <!-- Content: EMPTY -->
</requires>
<!-- Contained by: plugin -->
```

This element indicates that the plugin depends upon another plugin or requires an extension point. Normally implicit requirements are sufficient, but on occasion it is necessary to make requirements explicit. These are occasions when the required plugin or extension point will be used programmatically rather than through the plugin file. Additionally, since extensions are not processed until after the dependency analysis phase is complete, some extension points may require that extensions are explicit about which exports they make use of.

Exactly one of attributes `plugin` or `point` must be used.

If the requirement is for a plugin then a particular version may optionally be checked for.

9.1. Attribute plugin

This is the id of the required plugin.

This attribute must not be specified if `point` is also specified.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

9.2. Attribute point

This is the id of the required extension point.

This attribute must not be specified if `plugin` is also specified.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

9.3. Attribute version

This is the version of the required plugin.

This attribute must not be specified if `point` is also specified.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

9.4. Attribute match

This attribute describes the type of validation for the plugin version.

This attribute must not be specified if `point` is also specified.

This attribute is variable expanded. Variables are resolved in the context of the plugin.

10. Element typedef

```
<typedef
    name      = id
    signature = signature>
    <!-- Content: EMPTY -->
</typedef>
<!-- Contained by: plugin -->
```

10.1. Attribute name

This attribute is variable expanded. Variables are resolved in the context of the plugin.

10.2. Attribute signature

This attribute is variable expanded. Variables are resolved in the context of the plugin.