Ice.Plugin.*

On this page:

Ice.Plugin.*name*

Synopsis

`Ice.Plugin.`*`name`*`=`*`entry_point`* `[`*`args`*`]`

Description

Defines a plug-in to be installed during communicator initialization. The format of *`entry_point`* varies by Ice implementation language, therefore this property cannot be defined in a configuration file that is shared by programs in different languages. Ice provides an alternate syntax that facilitates such sharing:

- `Ice.Plugin.`*`name`*`.cpp` for C++
- `Ice.Plugin.`*`name`*`.java` for Java
- `Ice.Plugin.`*`name`*`.clr` for the .NET Common Language Runtime

Refer to the relevant property for your language mapping for details on the entry point syntax.

Back to Top ^

Ice.Plugin.*name*.clr

Synopsis

`Ice.Plugin.`*`name`*`.clr=`*`assembly`*`:`*`class`* `[`*`args`*`]`

Description

Defines a .NET plug-in to be installed during communicator initialization. The *`assembly`* component can be a partially or fully qualified assembly name, such as `myplugin,Version=0.0.0.0,Culture=neutral`, or an assembly DLL name such as `myplugin.dll` that may optionally include a leading relative or absolute path name.

The locations that are searched for the assembly varies depending on how you define the *`assembly`* component:

Value for *`assembly`*	Example	Semantics

Fully-qualified assembly name (strong-named assembly)	`myplugin,Version=...,`
	`Culture=neutral,`
	`publicKeyToken=...`
		1. Checks assemblies that have already been loaded
		2. Searches the Global Assembly Cache (GAC)
		3. Searches the directory containing the `iceboxnet` executable

Partially-qualified assembly name	`myplugin`	1. Checks assemblies that have already been loaded 2. Searches the directory containing the `iceboxnet` executable
Relative path name	`plugins\MyPlugin.dll`	Path name is relative to the application's current working directory. Be sure to include the `.dll` extension in the path name.
Absolute path name	`C:\plugins\MyPlugin.dll`	Assembly must reside at the specified path name. Be sure to include the `.dll` extension in the path name.

See MSDN for more information on how the CLR locates assemblies.

The specified *`class`* must implement the `Ice.PluginFactory` interface. Any arguments that follow the class name are passed to the factory's `create` method. For example:

```
Ice.Plugin.MyPlugin.clr=MyFactory,Version=1.2.3.4:MyFactory arg1 arg2
```

Whitespace separates the arguments, and any arguments that contain whitespace must be enclosed in quotes.

If you specify a relative path name in the entry point, the assembly is located relative to the program's current working directory:

```
Ice.Plugin.MyPlugin.clr=..\MyFactory.dll:MyFactory arg1 arg2
```

Enclose the assembly's path name in quotes if it contains spaces:

```
Ice.Plugin.MyPlugin.clr="C:\Program Files\MyPlugin\MyFactory.dll:MyFactory" arg1 arg2
```

Back to Top ^

Ice.Plugin.*name*.cpp

Synopsis

`Ice.Plugin.`*`name`*`.cpp=`*`path`*`[,`*`version`*`]:`*`function`* `[`*`args`*`]`

Description

Defines a C++ plug-in to be installed during communicator initialization. The *`path`* and optional *`version`* components are used to construct the path name of a DLL or shared library. If no version is supplied, the Ice version is used. The *`function`* component is the name of a function with C linkage. For example, the entry point `MyPlugin,37:create` would imply a shared library name of `libMyPlugin.so.37` on Unix and `MyPlugin37.dll` on Windows. Furthermore, if Ice is built on Windows with debugging, a d is automatically appended to the version (for example, `MyPlugin37d.dll`). The configuration is the same for the C++11 mapping and the C++98 mapping: Ice computes the name of the shared library to load and adds automatically a "++11" suffix when needed.

The function must be declared with external linkage and have the following signature:

C++11

```
Ice::Plugin* function(const std::shared_ptr<Ice::Communicator>& communicator,
                      const std::string& name,
                      const Ice::StringSeq& args);
```

C++98

```
Ice::Plugin* function(const Ice::CommunicatorPtr& communicator,
                      const std::string& name,
                      const Ice::StringSeq& args);
```

Note that the function must return a pointer and not a smart pointer.

Any arguments that follow the entry point are passed to the entry point function. For example:

```
Ice.Plugin.MyPlugin.cpp=MyFactory,37:create arg1 arg2
```

Whitespace separates the arguments, and any arguments that contain whitespace must be enclosed in quotes.

The *`path`* component may optionally contain a relative or absolute path name, indicated by the presence of a path separator (`/` or `\`). In this case, the last component of the path is used to construct the version-specific name of the shared library or DLL. Consider this example:

```
Ice.Plugin.MyPlugin.cpp=./MyFactory,37:create arg1 arg2
```

The use of a relative path means the Ice run time will look in the current working directory for `libMyPlugin.so.37` on Unix or `MyPlugin37.dll` on Windows.

If the *`path`* component contains spaces, the entire entry point must be enclosed in quotes:

```
Ice.Plugin.MyPlugin.cpp="C:\Program Files\MyPlugin\MyFactory,37:create" arg1 arg2
```

If the *path* component does not include a leading path name, Ice delegates to the operating system to locate the shared library or DLL, which typically means that the plug-in can reside in any of the directories in your shared library or DLL search path.

When the plug-in is packaged in a static library and linked into the application through `Ice::registerPluginFactory`, the entry point (*path*[, *version*]:*function)* component of this property is ignored. The `args`, if any, are preserved, and are given to the registered plug-in factory function when the plug-in is created.

Back to Top ^

Ice.Plugin.*name*.java

Synopsis

`Ice.Plugin.`*`name`*`.java=[`*`path`*`:]`*`class`* `[`*`args`*`]`

Description

Defines a Java plug-in to be installed during communicator initialization. The specified class must implement the `com.zeroc.Ice.PluginFactory` interface. Any arguments that follow the class name are passed to the `create` method. For example:

```
Ice.Plugin.MyPlugin.java=MyFactory arg1 arg2
```

Whitespace separates the arguments, and any arguments that contain whitespace must be enclosed in quotes.

If *`path`* is specified, it may be the path name of a JAR file or class directory, as shown below:

```
Ice.Plugin.MyPlugin.java=MyFactory.jar:MyFactory
Ice.Plugin.MyOtherPlugin.java=/classes:MyOtherFactory
```

If *`path`* contains spaces, it must be enclosed in quotes:

```
Ice.Plugin.MyPlugin.java="factory classes.jar":MyFactory
```

If *`class`* is specified without a path, Ice attempts to load the class using class loaders in a well-defined order.

Back to Top ^