Plug-in API

Previous

Next

On this page:

The `Plugin` Interface

The plug-in facility defines a local Slice interface that all plug-ins must implement:

Slice

```
module Ice
{
    local interface Plugin
    {
        void initialize();
        void destroy();
    }
}
```

The lifecycle of an Ice plug-in is structured to accommodate dependencies between plug-ins, such as when a logger plug-in needs to use IceSSL for its logging activities. Consequently, a plug-in object's lifecycle consists of four phases:

- Construction
  The Ice run time uses a language-specific factory API for instantiating plug-ins. During construction, a plug-in can acquire resources but must not spawn new threads or perform activities that depend on other plug-ins.

- Initialization
  After all plug-ins have been constructed, the Ice run time invokes `initialize` on each plug-in. The order in which plug-ins are initialized is undefined by default but can be customized using a configuration property. If a plug-in has a dependency on another plug-in, you must configure the Ice run time so that initialization occurs in the proper order. In this phase it is safe for a plug-in to spawn new threads; it is also safe for a plug-in to interact with other plug-ins and use their services, as long as those plug-ins have already been initialized. If `initialize` raises an exception, the Ice run time invokes `destroy` on all plug-ins that were successfully initialized (in the reverse order of initialization) and raises the original exception to the application.

- Active
  The active phase spans the time between initialization and destruction. Plug-ins must be designed to operate safely in the context of multiple threads.

- Destruction
  The Ice run time invokes `destroy` on each plug-in in the reverse order of initialization.

  A plug-in's `destroy` implementation must not make remote invocations.

This lifecycle is repeated for each new communicator that an application creates and destroys.

Back to Top ^

C++ Plug-in Factory

In C++, a plug-in factory is a function with C linkage and the following signature:

C++11

C++

```
extern "C"
{
    Ice::Plugin* functionName(const std::shared_ptr<Ice::Communicator>& communicator,
                              const std::string& name,
                              const Ice::StringSeq& args)
    {
       ...
    }
}
```

C++98

C++

```
extern "C"
{
    Ice::Plugin* functionName(const Ice::CommunicatorPtr& communicator,
                              const std::string& name,
                              const Ice::StringSeq& args)
    {
       ...
    }
}
```

You can choose any name for the factory function.

Since the function uses C linkage, it must return the plug-in object as a regular C++ pointer and not as a smart pointer. Furthermore, the function must not raise C++ exceptions; if an error occurs, the function must return zero. The arguments to the function consist of the communicator that is in the process of being initialized, the name assigned to the plug-in, and any arguments that were specified in the plug-in's configuration.

If your plug-in and the associated factory function are packaged in a shared library or DLL loaded at run time, you need to export this function from the shared library or DLL. We provide the macro `ICE_DECLSPEC_EXPORT` for this purpose:

C++

```
#if defined(_MSC_VER)
#   define ICE_DECLSPEC_EXPORT __declspec(dllexport)
...
#elif defined(__GNUC__) || defined(__clang__)
#   define ICE_DECLSPEC_EXPORT __attribute__((visibility ("default")))
...
```

Simply add `ICE_DECLSPEC_EXPORT` to the definition of your plug-in factory:

C++11

C++

```
extern "C"
{
    ICE_DECLSPEC_EXPORT Ice::Plugin* functionName(const std::shared_ptr<Ice::Communicator>& communicator,
                                                  const std::string& name,
                                                  const Ice::StringSeq& args)
    {
       ...
    }
}
```

C++98

C++

```
extern "C"
{
    ICE_DECLSPEC_EXPORT Ice::Plugin* functionName(const Ice::CommunicatorPtr& communicator,
                                                  const std::string& name,
                                                  const Ice::StringSeq& args)
    {
       ...
    }
}
```

If you don't want to rely on the dynamic loading of your plug-in shared library or DLL at run time, or if your plug-in is packaged in a static library, you can also link the plug-in into your application and call `Ice::registerPluginFactory` in your main application's code to register the plug-in before you initialize Ice communicators. For example:

C++

```
MyApp::MyApp()
{
   // Load/link the "IceSSL" plug-in before communicator initialization

   Ice::registerPluginFactory("IceSSL", createIceSSL, false);
}
```

The `registerPluginFactory` function registers the plug-in's factory function with the Ice run time. It returns `void`, and accepts the following parameters:

- `const string&`
  The name of the plug-in.
- `PLUGIN_FACTORY`
  A pointer to the plug-in factory function.
- `bool`
  When `true`, the plug-in is always loaded (created) during communicator initialization, even if `Ice.Plugin.`*`name`* is not set. When `false`, the plug-in is loaded (created) during communication initialization only if `Ice.Plugin.`*`name`* is set to a non-empty value (e.g.: `Ice.Plugin.IceSSL=1`).

Back to Top ^

Java Plug-in Factory

In Java, a plug-in factory must implement the `PluginFactory` interface:

Java

```
package com.zeroc.Ice;

public interface PluginFactory
{
    Plugin create(Communicator communicator, String name, String[] args);
}
```

Java Compat

```
package Ice;

public interface PluginFactory
{
    Plugin create(Communicator communicator, String name, String[] args);
}
```

The arguments to the `create` method consist of the communicator that is in the process of being initialized, the name assigned to the plug-in, and any arguments that were specified in the plug-in's configuration.

The `create` method can return `null` to indicate that a general error occurred, or it can raise `PluginInitializationException` to provide more detailed information. If any other exception is raised, the Ice run time wraps it inside an instance of `PluginInitializationException`.

Back to Top ^

C# Plug-in Factory

In .NET, a plug-in factory must implement the `Ice.PluginFactory` interface:

C#

```
namespace Ice
{
    public interface PluginFactory
    {
        Plugin create(Communicator communicator, string name, string[] args);
    }
}
```

The arguments to the `create` method consist of the communicator that is in the process of being initialized, the name assigned to the plug-in, and any arguments that were specified in the plug-in's configuration.

The `create` method can return `null` to indicate that a general error occurred, or it can raise `PluginInitializationException` to provide more detailed information. If any other exception is raised, the Ice run time wraps it inside an instance of `PluginInitializationException`.

Back to Top ^

See Also

- Plug-in Configuration
- Advanced Plug-in Topics

Previous

Next