

Advanced Plug-in Topics

This page discusses additional aspects of the Ice plug-in facility that may be of use to applications with special requirements.

On this page:

- [Plug-in Dependencies](#)
- [The Plug-in Manager](#)
- [Delayed Plug-in Initialization](#)

Plug-in Dependencies

If a plug-in has a dependency on another plug-in, you must ensure that Ice initializes the plug-ins in the proper order. Suppose that a custom logger implementation depends on [IceSSL](#); for example, the logger may need to transmit log messages securely to another server. We start with the following C++ configuration:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
Ice.Plugin.MyLogger=MyLogger:createMyLogger
```

The problem with this configuration is that it does not specify the order in which the plug-ins should be loaded and initialized. If the Ice run time happens to initialize `MyLogger` first, the plug-in's `initialize` method will fail if it attempts to use the services of the uninitialized `IceSSL` plug-in.

To remedy the situation, we need to add one more property:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
Ice.Plugin.MyLogger=MyLogger:createMyLogger
Ice.PluginLoadOrder=IceSSL, MyLogger
```

Using the [Ice.PluginLoadOrder](#) property we can guarantee that the plug-ins are loaded in the correct order.

The Plug-in Manager

`PluginManager` is the name of an internal Ice object that is responsible for managing all aspects of Ice plug-ins. This object supports a [local](#) Slice interface of the same name, and an application can obtain a reference to this object using the following communicator operation:

Slice

```
module Ice {
  local interface Communicator {
    PluginManager getPluginManager();
    // ...
  };
};
```

The `PluginManager` interface offers three operations:

Slice

```
module Ice {
  local interface PluginManager {
    void initializePlugins();
    Plugin getPlugin(string name);
    void addPlugin(string name, Plugin pi);
  };
};
```

The `initializePlugins` operation is used in special cases when an application needs to manually initialize one or more plug-ins, as discussed in the next section.

The `getPlugin` operation returns a reference to a specific plug-in. The `name` argument must match an installed plug-in, otherwise the operation raises `NotRegisteredException`. This operation is useful when a plug-in exports an interface that an application can use to query or customize its attributes or behavior.

Finally, `addPlugin` provides a way for an application to install a plug-in directly, without the use of a configuration property.

Delayed Plug-in Initialization

It is sometimes necessary for an application to manually configure a plug-in prior to its initialization. For example, SSL keys are often protected by a passphrase, but a developer may be understandably reluctant to specify that passphrase in a configuration file because it would be exposed in clear text. The developer would likely prefer to configure the `IceSSL` plug-in with a password callback instead; however, this must be done before the plug-in is initialized and attempts to load the SSL key. The solution is to configure the Ice run time so that it postpones the initialization of its plug-ins:

```
Ice.InitPlugins=0
```

When `Ice.InitPlugins` is set to zero, initializing plug-ins becomes the application's responsibility. The example below demonstrates how to perform this initialization:

C++

```
Ice::CommunicatorPtr ic = ...
Ice::PluginManagerPtr pm = ic->getPluginManager();
IceSSL::PluginPtr ssl = pm->getPlugin("IceSSL");
ssl->setPasswordPrompt(...);
pm->initializePlugins();
```

After obtaining the `IceSSL` plug-in and establishing the password callback, the application invokes `initializePlugins` on the plug-in manager object to commence plug-in initialization.

See Also

- [IceSSL](#)
- [Ice Plug-in Properties](#)