Logger Plug-ins

Previous Next

Installing a custom logger using the Ice plug-in facility has several advantages. Because the logger plug-in is specified by a configuration property and loaded dynamically by the Ice run time, an application requires no code changes in order to utilize a custom logger implementation. Furthermore, a logger plug-in takes precedence over the per-process logger and the logger supplied in the `InitializationData` argument during communicator initialization, meaning you can use a logger plug-in to override the logger that an application installs by default.

On this page:

Installing a C++ Logger Plug-in

To install a logger plug-in in C++, you must first define a subclass of `Ice::Logger`:

C++11

```
class MyLoggerI : public Ice::Logger
{
public:

    virtual void print(const std::string& message) override;
    virtual void trace(const std::string& category, const std::string& message) override;
    virtual void warning(const std::string& message) override;
    virtual void error(const std::string& message) override;
    virtual std::string getPrefix() override;
    virtual std::shared_ptr<Ice::Logger> cloneWithPrefix(const string& prefix) override;

    // ...
};
```

C++98

```
class MyLoggerI : public Ice::Logger
{
public:

    virtual void print(const std::string& message);
    virtual void trace(const std::string& category, const std::string& message);
    virtual void warning(const std::string& message);
    virtual void error(const std::string& message);
    virtual std::string getPrefix();
    virtual LoggerPtr cloneWithPrefix(const string& prefix);

    // ...
};
```

Next, supply a factory function that installs your custom logger by returning an instance of `Ice::LoggerPlugin`:

C++11

```
extern "C"
{
    ICE_DECLSPEC_EXPORT Ice::Plugin*
    createLogger(const std::shared_ptr<Ice::Communicator>& communicator,
                 const std::string& name,
                 const Ice::StringSeq& args)
    {
        auto logger = std::make_shared<MyLoggerI>();
        return new Ice::LoggerPlugin(communicator, logger);
    }
}
```

C++98

```
extern "C"
{
    ICE_DECLSPEC_EXPORT Ice::Plugin*
    createLogger(const Ice::CommunicatorPtr& communicator,
                 const std::string& name,
                 const Ice::StringSeq& args)
    {
        Ice::LoggerPtr logger = new MyLoggerI;
        return new Ice::LoggerPlugin(communicator, logger);
    }
}
```

The factory function can have any name you wish; we used `createLogger` in this example. Refer to the plug-in API for more information on plug-in factories.

The definition of `LoggerPlugin` is shown below:

C++11

```
namespace Ice
{
    class LoggerPlugin : public Plugin
    {
    public:
        LoggerPlugin(const std::shared_ptr<Communicator>&, const std::shared_ptr<Logger>&);

        virtual void initialize() override;
        virtual void destroy() override;
    }
}
```

C++98

```
namespace Ice
{
    class LoggerPlugin : public Plugin
    {
    public:
        LoggerPlugin(const CommunicatorPtr&, const LoggerPtr&);

        virtual void initialize();
        virtual void destroy();
    }
}
```

The constructor installs your logger into the given communicator. The `initialize` and `destroy` methods are empty, but you can subclass `LoggerPlugin` and override these methods if necessary.

Finally, define a configuration property that loads your plug-in into an application:

```
Ice.Plugin.MyLogger=mylogger:createLogger
```

The plug-in's name in this example is `MyLogger`; again, you can use any name you wish. The value of the property represents the plug-in's entry point, in which `mylogger` is the abbreviated form of its shared library or DLL, and `createLogger` is the name of the factory function.

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for C++ run time:

```
Ice.Plugin.MyLogger.cpp=mylogger:createLogger
```

Back to Top ^

Installing a Java Logger Plug-in

To install a logger plug-in in Java, you must first define a subclass of `Logger`:

Java

```
public class MyLoggerI implements com.zeroc.Ice.Logger
{
    public void print(String message) { ... }
    public void trace(String category, String message) { ... }
    public void warning(String message) { ... }
    public void error(String message) { ... }
    public String getPrefix() { ... }
    public Logger cloneWithPrefix(String prefix) { ... }

    // ...
}
```

Next, define a factory class that installs your custom logger by returning an instance of `LoggerPlugin`:

Java

```
public class MyLoggerPluginFactoryI implements com.zeroc.Ice.PluginFactory
{
    public com.zeroc.Ice.Plugin create(com.zeroc.Ice.Communicator communicator, String name, String[] args)
    {
        com.zeroc.Ice.Logger logger = new MyLoggerI();
        return new com.zeroc.Ice.LoggerPlugin(communicator, logger);
    }
}
```

The factory class can have any name you wish; in this example, we used `MyLoggerPluginFactoryI`. Refer to the plug-in API for more information on plug-in factories.

The definition of `LoggerPlugin` is shown below:

Java

```
package com.zeroc.Ice;

public class LoggerPlugin implements Plugin
{
    public LoggerPlugin(Communicator communicator, Logger logger) { ... }

    public void initialize() { }

    public void destroy() { }
}
```

Java Compat

```
package Ice;

public class LoggerPlugin implements Plugin
{
    public LoggerPlugin(Communicator communicator, Logger logger) { ... }

    public void initialize() { }

    public void destroy() { }
}
```

The constructor installs your logger into the given communicator. The `initialize` and `destroy` methods are empty, but you can subclass `LoggerPlugin` and override these methods if necessary.

Finally, define a configuration property that loads your plug-in into an application:

```
Ice.Plugin.MyLogger=MyLoggerPluginFactoryI
```

The plug-in's name in this example is `MyLogger`; again, you can use any name you wish. The value of the property is the name of the factory class.

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for Java run time:

```
Ice.Plugin.MyLogger.java=MyLoggerPluginFactoryI
```

Back to Top ^

Installing a C# Logger Plug-in

To install a logger plug-in in .NET, you must first define a subclass of `Ice.Logger`:

C#

```
public class MyLoggerI : Ice.Logger
{
    public void print(string message) { ... }
    public void trace(string category, string message) { ... }
    public void warning(string message) { ... }
    public void error(string message) { ... }
    public string getPrefix() { ... }
    public Logger cloneWithPrefix(string prefix) { ... }

    // ...
}
```

Next, define a factory class that installs your custom logger by returning an instance of `Ice.LoggerPlugin`:

C#

```
public class MyLoggerPluginFactoryI : Ice.PluginFactory
{
    public Ice.Plugin create(Ice.Communicator communicator, string name, string[] args)
    {
        Ice.Logger logger = new MyLoggerI();
        return new Ice.LoggerPlugin(communicator, logger);
    }
}
```

The factory class can have any name you wish; in this example, we used `MyLoggerPluginFactoryI`. Refer to the plug-in API for more information on plug-in factories. Typically the logger implementation and the factory are compiled into a single assembly.

The definition of `LoggerPlugin` is shown below:

C#

```
namespace Ice
{
    public partial class LoggerPlugin : Plugin
    {
        public LoggerPlugin(Communicator communicator, Logger logger)
        {
            // ...
        }

        public void initialize() { }

        public void destroy() { }
    }
}
```

The constructor installs your logger into the given communicator. The `initialize` and `destroy` methods are empty, but you can subclass `LoggerPlugin` and override these methods if necessary.

Finally, define a configuration property that loads your plug-in into an application:

```
Ice.Plugin.MyLogger=mylogger.dll:MyLoggerPluginFactoryI
```

The plug-in's name in this example is `MyLogger`; again, you can use any name you wish. The value of the property is the entry point for the factory, consisting of an assembly name followed by the name of the factory class.

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for .NET run time:

```
Ice.Plugin.MyLogger.clr=mylogger.dll:MyLoggerPluginFactoryI
```

Back to Top ^

See Also

- Custom Loggers
- The Per-Process Logger
- Communicator Initialization
- Plug-in API
- Ice.Plugin.*
- Ice.InitPlugins
- Ice.PluginLoadOrder

Previous Next