Configuring IceBox Services

On this page:

- Installing an IceBox Service
- IceBox Service Configuration in C++
- IceBox Service Configuration in Java
- IceBox Service Configuration in C#
- Load Order for IceBox Services
- Using a Shared Communicator
- Inheriting Properties from the IceBox Server
- Logging Considerations for IceBox Services

Installing an IceBox Service

A service is configured into an IceBox server using a single `IceBox.Service` property. This property serves several purposes: it defines the name of the service, it provides the server with the service entry point, and it defines properties and arguments for the service.

The format of the property is shown below:

```
IceBox.Service.name=entry_point [args]
```

The *name* component of the property key is the service name (IceStorm, in this example). This name is passed to the service's `start` operation, and must be unique among all services configured in the same IceBox server. It is possible, though rarely necessary, to load two or more instances of the same service under different names.

The first argument in the property value is the entry point specification. Any arguments following the entry point specification are examined. If an argument has the form `--`*`name`*`=`*`value`*, then it is interpreted as a property definition that appears in the property set of the communicator passed to the service `start` operation. These arguments are removed, and any remaining arguments are passed to the `start` operation in the `args` parameter.

IceBox Service Configuration in C++

For a C++ service, the entry point must have the form *library[,version]:symbol*, where *library* is the simple name of the service's shared library or DLL, and *symbol* is the name of the entry point function. A "simple name" is one without any platform-specific prefixes or extensions; the server adds appropriate decorations depending on the platform. The simple name may include a leading path, and the version is optional. If specified, the version is embedded in the library name.

As an example, here is how we could configure IceStorm, which is implemented as an IceBox service in C++:

```
IceBox.Service.IceStorm=IceStormService,35:createIceStorm
```

IceBox uses the information provided in the entry point specification to compose a library name. For the IceStorm example shown above, IceBox on Windows would compose the library name `IceStormService35.dll`. If IceBox is compiled with debug information, it appends a d to the library name, so the name becomes `IceStormService35d.dll` instead.

The exact name of the library that is loaded depends on the naming conventions of the platform IceBox executes on. For example, on an OS X machine, the library name is `libIceStormService35.dylib`.

If the simple name does not include a leading path, the shared library or DLL must reside in a directory that appears in `PATH` on Windows or the shared library search path (such as `LD_LIBRARY_PATH`) on POSIX systems.

The entry point function, *symbol*, must have the signature that we originally presented in our example:

C++

```
extern "C" IceBox::Service* function(Ice::CommunicatorPtr);
```

The communicator instance passed to this function is the IceBox server's communicator and should only be used for administrative purposes. For example, the entry point function could use this communicator's logger to display log messages. For a service's normal operations, it must use the communicator that it receives as an argument to its `start` method.

Here is a sample configuration for our C++ service:

```
IceBox.Service.Hello=HelloService:create --Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in `HelloService.dll` on Windows or `libHelloService.so` on Linux, and the entry point function `create` is invoked to create an instance of the service. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

IceBox Service Configuration in Java

For a Java service, the entry point is typically the class name (including any package) of the service implementation class, but may also include a leading path to a class directory or JAR file. The class must define a public constructor.

To instantiate the service, the IceBox server first checks to see if the service defines a constructor taking an argument of type `Ice.Communicator`. If so, the service invokes this constructor and passes the server's communicator, which should only be used for administrative purposes. For example, the constructor could use this communicator's logger to display log messages. For a service's normal operations, it must use the communicator that it receives as an argument to its `start` method.

If the service does not define a constructor taking an `Ice.Communicator` argument, the server invokes the service's default constructor.

Here is a sample configuration for our Java example:

```
IceBox.Service.Hello=HelloServiceI --Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in the class `HelloServiceI`. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

IceBox Service Configuration in C#

The entry point of a .NET service has the form *`assembly:class`*. The *`assembly`* component can be a partially or fully qualified assembly name, such as `hello,Version=0.0.0.0,Culture=neutral`, or an assembly DLL name such as `hello.dll` that may optionally include a leading relative or absolute path name.

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

Value for ***`assembly`***	Example	Semantics

Fully-qualified assembly name (strong-named assembly)	`hello,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	`hello`	1. Checks assemblies that have already been loaded 2. Searches the directory containing the `iceboxnet` executable
Relative path name	`services\MyService.dll`	Path name is relative to `iceboxnet`'s current working directory. Be sure to include the `.dll` extension in the path name.
Absolute path name	`C:\services\MyService.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 *`class`* component is the complete class name of the service implementation class, which must define a public constructor.

To instantiate the service, the IceBox server first checks to see if the service defines a constructor taking an argument of type `Ice.Communicator`. If so, the service invokes this constructor and passes the server's communicator, which should only be used for administrative purposes. For example, the constructor could use this communicator's logger to display log messages. For a service's normal operations, it must use the communicator that it receives as an argument to its `start` method.

If the service does not define a constructor taking an `Ice.Communicator` argument, the server invokes the service's default constructor.

Here is a sample configuration for our C# example:

```
IceBox.Service.Hello=helloservice.dll:HelloServiceI --Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in the assembly named `helloservice.dll`, implemented by the class `HelloServiceI`. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

Load Order for IceBox Services

By default, the server loads the configured services in an undefined order, meaning services in the same IceBox server should not depend on one another. If services must be loaded in a particular order, the `IceBox.LoadOrder` property can be used:

```
IceBox.LoadOrder=Service1,Service2
```

In this example, `Service1` is loaded first, followed by `Service2`. Any remaining services are loaded after `Service2`, in an undefined order. Each service mentioned in `IceBox.LoadOrder` must have a matching `IceBox.Service` property.

During shutdown, services are stopped in the reverse of the order in which they were loaded.

Using a Shared Communicator

IceBox creates a separate communicator instance for each service by default in order to minimize the chances of accidental conflicts among services. You can optionally specify that certain services use a shared communicator instead by setting `IceBox.UseSharedCommunicator.`*`name`* properties in the server's configuration:

```
IceBox.Service.Hello=...
IceBox.Service.Printer=...
IceBox.UseSharedCommunicator.Hello=1
IceBox.UseSharedCommunicator.Printer=1
```

A common use case for sharing a communicator between two or more services is enabling the use of collocation optimizations for invocations among those services. This optimization is not possible with the default behavior that creates a new communicator for each service.

IceBox prepares the property set of this shared communicator as follows:

- If services inherit the server's properties, the property set initially contains the server's properties (excluding `Ice.Admin.Endpoints`), otherwise the property set starts out empty.
- For each service that uses the shared communicator:
  - Merge its properties into the shared property set, overwriting any existing properties with the same names
  - Remove any properties from the shared property set that the service explicitly clears. For example, the definition
    `Ice.Trace.Network=`
    clears any existing setting of `Ice.Trace.Network` in the shared property set.
  - Translate service-specific command-line settings into properties (e.g., `--Hello.Debug=1`)

Service properties are merged in the same order as the services are loaded. As a result, the final value of a property that is defined by multiple services depends on the order in which those services are loaded. Let's expand our example to demonstrate this behavior:

```
# File: server.cfg
IceBox.Service.Hello=... --Ice.Config=hello.cfg --Hello.Debug=1
IceBox.Service.Printer=... --Ice.Config=printer.cfg
IceBox.UseSharedCommunicator.Hello=1
IceBox.UseSharedCommunicator.Printer=1
IceBox.InheritProperties=1
IceBox.LoadOrder=Hello,Printer
Ice.Trace.Network=1

# File: hello.cfg
Ice.Trace.Network=2

# File: printer.cfg
Ice.Trace.Network=3
```

The two services `Hello` and `Printer` use the shared communicator, and all services inherit the server's properties. As IceBox prepares the shared property set, the initial value of `Ice.Trace.Network` is `1` as defined in the (inherited) server's configuration. The `IceBox.LoadOrder` property specifies that the `Hello` service should be loaded first, therefore the `Ice.Trace.Network` property momentarily has the value 2 but ultimately has the value 3 after the properties for the `Printer` service are merged.

If we change the value of `IceBox.LoadOrder` so that IceBox loads `Printer` first, the value for `Ice.Trace.Network` in the shared communicator will be 2 instead because the setting in `hello.cfg` overrides all previous values.

Inheriting Properties from the IceBox Server

By default, a service does not inherit the IceBox server's configuration properties. For example, consider the following server configuration:

```
IceBox.Service.Weather=... --Ice.Config=svc.cfg
Ice.Trace.Network=1
```

The `Weather` service only receives the properties that are defined in its `IceBox.Service` property. In the example above, the service's communicator is initialized with the properties from the file `svc.cfg`.

If services need to inherit the IceBox server's configuration properties, define the `IceBox.InheritProperties` property in the IceBox server's configuration:

```
IceBox.Service.Weather=... --Ice.Config=svc.cfg
Ice.Trace.Network=1
IceBox.InheritProperties=1
```

All services inherit the server's properties when `IceBox.InheritProperties` is set to a non-zero value.

The properties of the shared communicator are also affected by this setting.

Logging Considerations for IceBox Services

In Ice 3.4 and earlier, the IceBox server configures a variation of its own logger in the communicator that it creates for each service (the only difference being a service-specific logging prefix), therefore the logging scheme you configure for the IceBox server must be appropriate for all of its services. The only way a service can configure a different logger is by using a logger plug-in.

As of Ice 3.5, the IceBox server only configures a logger for a service if that service has not already specified its own logger via the Ice.LogFile or Ice.UseSyslog properties.

See Also

- IceBox Properties
- Developing IceBox Services

- IceStorm
- Location Transparency
- Ice Miscellaneous Properties