Configuring IceBox Services

On this page:

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 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,34: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 `IceStormService34.dll`. If IceBox is compiled with debug information, it appends a d to the library name, so the name becomes `IceStormService34d.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 Apple machines, the library name is `libIceStormService34.dylib`.

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 simply the complete class name (including any package) of the service implementation class. The class must reside in the class path of the server, and 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 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.

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 specified as the name of a DLL present in `PATH`, or as the full name of an assembly residing in the Global Assembly Cache (GAC), such as `hello,Version=0.0.0.0,Culture=neutral`. 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 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.

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.

Using a Shared Communicator

A service can be configured to use a shared communicator using the `IceBox.UseSharedCommunicator.`*`name`* property:

```
IceBox.UseSharedCommunicator.Hello=1
```

The default behavior if this property is not specified is to create a new communicator instance for the service. However, if collocation optimizations between services are desired, each of those services must be configured to use the shared communicator.

Inheriting Properties from the IceBox Server

By default, a service does not inherit the 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 server's configuration properties, define the `IceBox.InheritProperties` property in the IceBox server's configuration:

```
IceBox.InheritProperties=1
```

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

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.

Logging Considerations for IceBox Services

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.

See Also

- IceBox Properties
- Developing IceBox Services
- IceStorm
- Location Transparency