Developing IceBox Services

On this page:

The IceBox `Service` Interface

Writing an IceBox service requires implementing the IceBox `Service` interface:

Slice

```
module IceBox
{
    local interface Service
    {
        void start(string name, Ice::Communicator communicator, Ice::StringSeq args);
        void stop();
    }
}
```

As you can see, a service needs to implement only two operations, `start` and `stop`. These operations are invoked by the server; `start` is called after the service is loaded, and `stop` is called when the IceBox server is shutting down.

The `start` operation is the service's opportunity to initialize itself; this typically includes creating an object adapter and servants. The `name` and `args` parameters supply information from the service's configuration, and the `communicator` parameter is an `Ice::Communicator` object created by the server for use by the service. Depending on the service configuration, this communicator instance may be shared by other services in the same IceBox server, therefore care should be taken to ensure that items such as object adapters are given unique names.

The `stop` operation must reclaim any resources used by the service. Generally, a service deactivates its object adapter, and may also need to invoke `waitForDeactivate` on the object adapter in order to ensure that all pending requests have been completed before the clean up process can proceed. The server is responsible for destroying the communicator instance that was passed to `start`.

Whether the service's implementation of `stop` should explicitly destroy its object adapter depends on other factors. For example, the adapter should be destroyed if the service uses a shared communicator, especially if the service could eventually be restarted. In other circumstances, the service can allow its adapter to be destroyed as part of the communicator's destruction.

These interfaces are declared as `local` for a reason: they represent a contract between the server and the service, and are not intended to be used by remote clients. Any interaction the service has with remote clients is done via servants created by the service.

Back to Top ^

IceBox Service Example in C++

The example we present here is taken from the `IceBox/hello` sample program provided in the Ice distribution.

The class definition for our service is quite straightforward:

C++11

```
#include <IceBox/IceBox.h>

class HelloServiceI : public IceBox::Service
{
public:

    virtual void start(const std::string&, const std::shared_ptr<Ice::Communicator>&, const Ice::StringSeq&)
override;
    virtual void stop() override;

private:

    std::shared_ptr<Ice::ObjectAdapter> _adapter;
};
```

C++98

```
#include <IceBox/IceBox.h>

class HelloServiceI : public IceBox::Service
{
public:

    virtual void start(const std::string&, const Ice::CommunicatorPtr&, const Ice::StringSeq&);
    virtual void stop();

private:

    Ice::ObjectAdapterPtr _adapter;
};
```

The member definitions are equally straightforward:

C++11

```
#include <Ice/Ice.h>
#include <HelloServiceI.h>
#include <HelloI.h>

using namespace std;

void
HelloServiceI::start(const string& name,
                     const shared_ptr<Ice::Communicator>& communicator,
                     const Ice::StringSeq& args)
{
    _adapter = communicator->createObjectAdapter(name);
    auto object = make_shared<HelloI>(communicator);
    _adapter->add(object, Ice::stringToIdentity("hello"));
    _adapter->activate();
}

void
HelloServiceI::stop()
{
    _adapter->deactivate();
}
```

C++98

```
#include <Ice/Ice.h>
#include <HelloServiceI.h>
#include <HelloI.h>

using namespace std;

void
HelloServiceI::start(const string& name,
                     const Ice::CommunicatorPtr& communicator,
                     const Ice::StringSeq& args)
{
    _adapter = communicator->createObjectAdapter(name);
    Ice::ObjectPtr object = new HelloI(communicator);
    _adapter->add(object, Ice::stringToIdentity("hello"));
    _adapter->activate();
}

void
HelloServiceI::stop()
{
    _adapter->deactivate();
}
```

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

C++ Service Entry Point

The last piece of the puzzle is the *entry point* function, which the IceBox server calls to obtain an instance of the service:

C++11

```
extern "C"
{
    ICE_DECLSPEC_EXPORT IceBox::Service*
    create(const shared_ptr<Ice::Communicator>&)
    {
        return new HelloServiceI;
    }
}
```

C++98

```
extern "C"
{
    ICE_DECLSPEC_EXPORT IceBox::Service*
    create(Ice::CommunicatorPtr)
    {
        return new HelloServiceI;
    }
}
```

In this example, the `create` function returns a new instance of the `Hello` service. The name of the function is not important, but it must have the signature shown above. In particular, the function must have C linkage, accept a single parameter (a `const std::shared_ptr<Ice::Communicator>&` in C++ and a `Ice::CommunicatorPtr` in C++98) and return a native pointer to `IceBox::Service`.

Configuring IceBox Services provides more information on entry points and describes how to configure your service into an IceBox server.

IceBox Service Example in Java

As with the C++ example presented in the previous section, the complete source for the Java example can be found in the `IceBox/hello` directory of the Ice distribution. The class definition for our service looks as follows:

Java

```
public class HelloServiceI implements com.zeroc.IceBox.Service
{
    public void start(String name, com.zeroc.Ice.Communicator communicator, String[] args)
    {
        _adapter = communicator.createObjectAdapter(name);
        com.zeroc.Ice.Object object = new HelloI(communicator);
        _adapter.add(object, com.zeroc.Ice.Util.stringToIdentity("hello"));
        _adapter.activate();
    }

    public void stop()
    {
        _adapter.deactivate();
    }

    private com.zeroc.Ice.ObjectAdapter _adapter;
}
```

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

The server requires a service implementation to provide a public constructor (a default constructor or a constructor that accepts an Ice communicator parameter). This is the *entry point* for a Java IceBox service; that is, the server dynamically loads the service implementation class and invokes this public constructor to obtain an instance of the service.

This example is a trivial service, and yours will likely be much more interesting, but this does demonstrate how easy it is to write an IceBox service. After compiling the service implementation class, it can be configured into an IceBox server as described in Configuring IceBox Services.

Back to Top ^

IceBox Service Example in C#

The complete source for the C# example can be found in the `IceBox/hello` directory of the Ice distribution. The class definition for our service looks as follows:

C#

```
class HelloServiceI : IceBox.Service
{
    public void start(string name, Ice.Communicator communicator, string[] args)
    {
        _adapter = communicator.createObjectAdapter(name);
        _adapter.add(new HelloI(), Ice.Util.stringToIdentity("hello"));
        _adapter.activate();
    }

    public void stop()
    {
        _adapter.deactivate();
    }

    private Ice.ObjectAdapter _adapter;
}
```

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

The server requires a service implementation to have a public constructor (a default constructor or a constructor that accepts an Ice communicator parameter). This is the entry point for a C# IceBox service; that is, the server dynamically loads the service implementation class from an assembly and invokes this public constructor to obtain an instance of the service.

This example is a trivial service, and yours will likely be much more interesting, but this does demonstrate how easy it is to write an IceBox service. After compiling the service implementation class, it can be configured into an IceBox server as described in Configuring IceBox Services.

Back to Top ^

IceBox Service Failures

An exception raised by a service's implementation of its entry point, `start`, or `stop` methods causes IceBox to log a message. An exception that occurs during server startup also results in [server termination](#).

A service implementation can indicate a failure by raising `IceBox::FailureException`:

Slice

```
module IceBox
{
    local exception FailureException
    {
        string reason;
    }
}
```

Note that, as a local exception, C++ users must instantiate `FailureException` with file and line number information:

C++

```
throw IceBox::FailureException(__FILE__, __LINE__, "my error message");
```

[Back to Top ^](#)

See Also

- [Configuring IceBox Services](#)
- [Starting the IceBox Server](#)