Writing an Ice Application with Ruby

Previous

This page shows how to create an Ice client application with Ruby.

On this page:

Compiling a Slice Definition for Ruby

The first step in creating our Ruby application is to compile our Slice definition to generate Ruby proxies. You can compile the definition as follows:

```
$ slice2rb Printer.ice
```

The `slice2rb` compiler produces a single source file, `Printer.rb`, from this definition. The exact contents of the source file do not concern us for now — it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Back to Top ^

Writing a Client in Ruby

The client code, in `Client.rb`, is shown below in full:

Ruby

```ruby
require 'Printer.rb'

begin
    communicator = Ice::initialize(ARGV)
    base = communicator.stringToProxy("SimplePrinter:default -h localhost -p 10000")
    printer = Demo::PrinterPrx::checkedCast(base)
    if not printer
        raise "Invalid proxy"
    end

    printer.printString("Hello World!")
ensure
    if defined? communicator and communicator != nil
        communicator.destroy()
    end
end
```

The program begins with a `require` statement, which loads the Ruby code we generated from our Slice definition in the previous section. It is not necessary for the client to explicitly load the `Ice` module because `Printer.rb` does that for you.

The body of the main program contains a `begin` block in which we place all the client code, followed by an `ensure` block.

The body of our `begin` block goes through the following steps:

1. We initialize the Ice run time by calling `Ice::initialize`. (We pass `ARGV` to this call because the client may have command-line arguments that are of interest to the run time; for this example, the client does not require any command-line arguments.) The call to `initialize` returns an `Ice::Communicator` reference, which is the main object in the Ice run time.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo::PrinterPrx::checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::`

`Printer` interface?" If so, the call returns a proxy of type `Demo::PrinterPrx`; otherwise, if the proxy denotes an interface of some other type, the call returns `nil`.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Before the code exits, the `ensure` block destroys the communicator, if one was created successfully. Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results.

Back to Top ^

Running the Client in Ruby

The server must be started before the client. Since Ice for Ruby does not support server-side behavior, we need to use a server from another language mapping. In this case, we will use the C++ server:

```
$ server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ ruby Client.rb
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
exception ::Ice::ConnectionRefusedException
{
    error = 111
}
```

Note that, to successfully run the client, the Ruby interpreter must be able to locate the Ice extension for Ruby. See the Ice for Ruby installation instructions for more information.

Back to Top ^

See Also

- Client-Side Slice-to-Ruby Mapping
- IceGrid

Previous