Writing an Ice Application with Visual Basic

This page shows how to create an Ice application with Visual Basic.

On this page:

Visual Basic Development Process

As of version 3.3, Ice no longer includes a separate compiler to create Visual Basic source code from Slice definitions. Instead, you need to use the Slice-to-C# compiler `slice2cs` to create C# source code and compile the generated C# source code with a C# compiler into a DLL that contains the compiled generated code for your Slice definitions. Your Visual Basic application then links with this DLL and the Ice for .NET DLL (`Ice.dll`).

> ⓘ This approach works not only with Visual Basic, but with any language that targets the .NET run time. However, ZeroC does not provide support for languages other than C# and Visual Basic.

The following illustration demonstrates this development process:
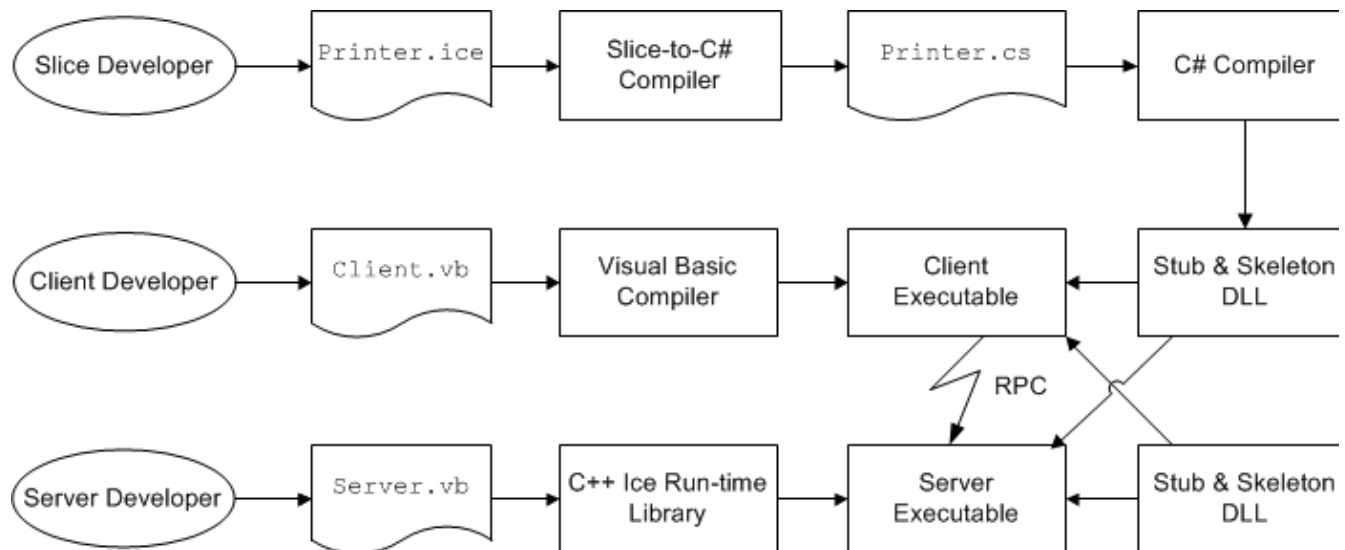
Developing a Visual Basic application with Ice.

Compiling a Slice Definition for Visual Basic

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

```
> mkdir generated
> slice2cs --output-dir generated Printer.ice
```

The `--output-dir` option instructs the compiler to place the generated files into the `generated` directory. This avoids cluttering the working directory with the generated files. The `slice2cs` compiler produces a single source file, `Printer.cs`, from this definition. The exact contents of this file do not concern us for now — it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

We now need to compile this generated code into a DLL:

```
> csc /reference:Ice.dll /lib:%ICE_HOME%\bin /t:library /out:Printer.dll generated\Printer.cs
```

This creates a DLL called `Printer.dll` that contains the code we generated from the Slice definitions.

Writing and Compiling a Server in Visual Basic

To implement our `Printer` interface, we must create a servant class. By convention, a servant class uses the name of its interface with an `I`-suffix, so our servant class is called `PrinterI` and placed into a source file `Server.vb`:

Visual Basic

```
Imports System
Imports Demo

Public Class PrinterI
    Inherits PrinterDisp_

    Public Overloads Overrides Sub printString( _
                             ByVal s As String, _
                             ByVal current As Ice.Current)
        Console.WriteLine(s)
    End Sub

End Class
```

The `PrinterI` class inherits from a base class called `PrinterDisp_`, which is generated by the `slice2cs` compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.Current` parameter.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code follows in `Server.vb` and is shown in full here:

Visual Basic

```
Module Server

    Public Sub Main(ByVal args() As String)

        Dim status As Integer = 0
        Dim ic As Ice.Communicator = Nothing
        Try
            ic = Ice.Util.initialize(args)
            Dim adapter As Ice.ObjectAdapter = _
                ic.createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -p 10000")
            Dim obj As Ice.Object = New PrinterI
            adapter.add(obj, ic.stringToIdentity("SimplePrinter"))
            adapter.activate()
            ic.waitForShutdown()
        Catch e As Exception
            Console.Error.WriteLine(e)
            status = 1
        End Try
        If Not ic Is Nothing Then
            ' Clean up
            '
            Try
                ic.destroy()
            Catch e As Exception
                Console.Error.WriteLine(e)
                status = 1
            End Try
        End If
        Environment.Exit(status)
    End Sub

End module
```

Note the general structure of the code:

Visual Basic

```
Module Server

    Public Sub Main(ByVal args() As String)

        Dim status As Integer = 0
        Dim ic As Ice.Communicator = Nothing
        Try

            ' Server implementation here...

        Catch e As Exception
            Console.Error.WriteLine(e)
            status = 1
        End Try
        If Not ic Is Nothing Then
            ' Clean up
            '
            Try
                ic.destroy()
            Catch e As Exception
                Console.Error.WriteLine(e)
                status = 1
            End Try
        End If
        Environment.Exit(status)
    End Sub

End module
```

The body of `Main` contains a `Try` block in which we place all the server code, followed by a `Catch` block. The catch block catches all exceptions that may be thrown by the code; the intent is that, if the code encounters an unexpected run-time exception anywhere, the stack is unwound all the way back to `Main`, which prints the exception and then returns failure to the operating system.

Before the code exits, it 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.

The body of our `Try` block contains the actual server code:

Visual Basic

```
            ic = Ice.Util.initialize(args)
            Dim adapter As Ice.ObjectAdapter = _
                ic.createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -p 10000")
            Dim obj As Ice.Object = New PrinterI
            adapter.add(obj, ic.stringToIdentity("SimplePrinter"))
            adapter.activate()
            ic.waitForShutdown()
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.Util.initialize`. (We pass `args` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server 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. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the servant. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)

6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. (Ice provides such a helper class, called `Ice.Application`.) As far as actual application code is concerned, the server contains only a few lines: ten lines for the definition of the `PrinterI` class, plus three lines to instantiate a `PrinterI` object and register it with the object adapter.

We can compile the server code as follows:

```
> vbc /reference:Ice.dll /libpath:%ICE_HOME%\bin /reference:Printer.dll /out:server.exe Server.vb
```

This compiles our application code and links it with the Ice run time and the DLL we generated earlier. We assume that the `ICE_HOME` environment variable is set to the top-level directory containing the Ice run time. (For example, if you have installed Ice in `C:\Ice`, set `ICE_HOME` to that path.)

Writing and Compiling a Client in Visual Basic

The client code, in `Client.vb`, looks very similar to the server. Here it is in full:

Visual Basic

```vb
Imports System
Imports Demo

Module Client

    Public Sub Main(ByVal args() As String)
        Dim status As Integer = 0
        Dim ic As Ice.Communicator = Nothing
        Try
            ic = Ice.Util.initialize(args)
            Dim obj As Ice.ObjectPrx = ic.stringToProxy("SimplePrinter:default -p 10000")
            Dim printer As PrinterPrx = PrinterPrxHelper.checkedCast(obj)
            If printer Is Nothing Then
                Throw New ApplicationException("Invalid proxy")
            End If

            printer.printString("Hello World!")
        Catch e As Exception
            Console.Error.WriteLine(e)
            status = 1
        End Try
        If Not ic Is Nothing Then
            ' Clean up
            '
            Try
                ic.destroy()
            Catch e As Exception
                Console.Error.WriteLine(e)
                status = 1
            End Try
        End If
        Environment.Exit(status)
    End Sub

End Module
```

Note that the overall code layout is the same as for the server: we use the same `Try` and `Catch` blocks to deal with errors. The code in the `Try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice.Util.initialize`.
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 `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrxHelper.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy of type `Demo::Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns null.

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.

Compiling the client looks much the same as for the server:

```
> vbc /reference:Ice.dll /libpath:%ICE_HOME%\bin /reference:Printer.dll /out:client.exe Client.vb
```

Running Client and Server in Visual Basic

To run client and server, we first start the server in a separate window:

```
> server.exe
```

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:

```
> client.exe
>
```

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 for now. (We will see cleaner ways to terminate a server in our discussion of `Ice.Application`.)

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:

```
Ice.ConnectionRefusedException
    error = 0
   at IceInternal.ProxyFactory.checkRetryAfterException(LocalException ex, Reference ref, Int32 cnt)
   at Ice.ObjectPrxHelperBase.handleException__(ObjectDel_ delegate, LocalException ex, Int32 cnt)
   at Ice.ObjectPrxHelperBase.ice_isA(String id__, Dictionary`2 context__, Boolean explicitContext__)
   at Ice.ObjectPrxHelperBase.ice_isA(String id__)
   at Demo.PrinterPrxHelper.checkedCast(ObjectPrx b)
   at Client.Main(String[] args)
Caused by: System.ComponentModel.
Win32Exception: No connection could be made because the target machine actively refused
 it
```

Note that, to successfully run client and server, the VB run time must be able to locate the `Ice.dll` library. (Under Windows, one way to ensure this is to copy the library into the current directory. Please consult the documentation for your VB run time to see how it locates libraries.)

See Also

- Client-Side Slice-to-C-Sharp Mapping
- Server-Side Slice-to-C-Sharp Mapping
- The `Ice.Application` Class
- The Current Object
- IceGrid