Server-Side C-Sharp Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing methods in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

On this page:

Skeleton Classes in C#

On the client side, interfaces map to proxy classes. On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has an abstract method for each operation on the corresponding interface. For example, consider our Slice definition for the `Node` interface:

Slice

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The Slice compiler generates the following definition for this interface:

C#

```
namespace Filesystem
{
    public interface NodeOperations_
    {
        string name(Ice.Current __current);
    }

    public interface NodeOperationsNC_
    {
        string name();
    }

    public interface Node : Ice.Object, NodeOperations_, NodeOperationsNC_
    {
    }

    public abstract class NodeDisp_ : Ice.ObjectImpl, Node
    {
        public string name()
        {
            return name(new Ice.Current());
        }

        public abstract string name(Ice.Current __current);

        // Mapping-internal code here...
    }
}
```

The important points to note here are:

- As for the client side, Slice modules are mapped to C# namespaces with the same name, so the skeleton class definitions are part of the Fi lesystem namespace.
- For each Slice interface <*interface-name*>, the compiler generates C# interfaces <*interface-name*>Operations_ and <*interface -name*>OperationsNC_ (NodeOperations_ and NodeOperationsNC_ in this example). These interfaces contain a method for each operation in the Slice interface. (You can ignore the Ice.Current parameter for the now.)
- For each Slice interface <*interface-name*>, the compiler generates a C# interface <*interface-name*> (Node in this example). That interface extends Ice.Object and the two operations interfaces.
- For each Slice interface <*interface-name*>, the compiler generates an abstract class <*interface-name*>Disp_ (NodeDisp_ in this example). This abstract class is the actual skeleton class; it is the base class from which you derive your servant class.

Servant Classes in C#

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the Node interface, you could write:

C#

```
public class NodeI : NodeDisp_
{
    public NodeI(string name)
    {
        _name = name;
    }

    public override string name(Ice.Current current)
    {
        return _name;
    }

    private string _name;
}
```

By convention, servant classes have the name of their interface with an I-suffix, so the servant class for the Node interface is called NodeI. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.) Note that NodeI extends N odeDisp_, that is, it derives from its skeleton class.

As far as Ice is concerned, the NodeI class must implement only a single method: the abstract name method that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other methods and data members as you see fit to support your implementation. For example, in the preceding definition, we added a _name member and a constructor. (Obviously, the constructor initializes the _na me member and the name method returns its value.)

Server-Side Normal and idempotent Operations in C#

Whether an operation is an ordinary operation or an idempotent operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

Slice

```
interface Example {
   void             normalOp();
   idempotent void   idempotentOp();
};
```

The operations class for this interface looks like this:

C#

```
public interface ExampleOperations_
{
    void normalOp(Ice.Current __current);
    void idempotentOp(Ice.Current __current);
}
```

Note that the signatures of the methods are unaffected by the `idempotent` qualifier.

See Also

- Slice for a Simple File System
- Parameter Passing in C-Sharp
- Raising Exceptions in C-Sharp
- Tie Classes in C-Sharp
- The Current Object