Server-Side Objective-C 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 Objective-C

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

Slice

```
["objc:prefix:FS"]
module Filesystem {
    interface Node {
        idempotent string name();
    };
// ...
};
```

The Slice compiler generates the following definition for this interface:

Objective-C

```
@protocol FSNode <ICEObject>
-(NSString *) name:(ICECurrent *)current;
@end

@interface FSNode : ICEObject
// ...
@end
```

As you can see, the server-side API consists of a protocol and a class, known as the *skeleton protocol* and *skeleton class*. The methods of the skeleton class are internal to the mapping, so they do not concern us here. The skeleton protocol defines one method for each Slice operation. As for the client-side mapping, the method name is the same as the name of the corresponding Slice operation. If the Slice operation has parameters or a return value, these are reflected in the generated method, just as they are for the client-side mapping. In addition, each method has an additional trailing parameter of type `ICECurrent`. This parameter provides additional information about an invocation to your server-side code.

As for the client-side mapping, the generated code reflects the fact that all Slice interfaces and classes ultimately derive from `Ice::Object`. As you can see, the generated protocol incorporates the `ICEObject` protocol, and the generated class derives from the `ICEObject` class.

Servant Classes in Objective-C

The Objective-C mapping supports two different ways to implement servants. You can implement a servant by deriving from the skeleton class and implementing the methods for the Slice operations in your derived class. Alternatively, you can use a delegate servant, which need not derive from the skeleton class.

Derived Servants in Objective-C

To provide an implementation for an Ice object, you can create a servant class that derives from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

Objective-C

```
@interface NodeI : FSNode <FSNode>
{
    @private
        NSString *myName;
}

+(id) nodei:(NSString *)name;
@end
```

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` derives from `FSNode`, that is, it derives from its skeleton class. In addition, it adopts the `FSNode` protocol. Adopting the protocol is not strictly necessary; however, if you do write your servants this way, the compiler emits a warning if you forget to implement one or more Slice operations for the corresponding interface, so we suggest that you make it a habit to always have your servant class adopt its skeleton protocol.

As far as Ice is concerned, the `NodeI` class must implement the single `name` method that is defined by its skeleton protocol. That way, the Ice run time gets a servant that can respond to the operation that is defined by its Slice interface. You can add other methods and instance variables as you see fit to support your implementation. For example, in the preceding definition, we added a `myName` instance variable and property, a convenience constructor, and `dealloc`. Not surprisingly, the convenience constructor initializes the `myName` instance variable, the `name` method returns the value of that variable, and `dealloc` releases it:

Objective-C

```
@implementation NodeI

+(id) nodei:(NSString *)name
{
    NodeI *instance = [[[NodeI alloc] init] autorelease];
    instance.myName = [[name copy] retain];
    return instance;
}

-(NSString *) name:(ICECurrent *)current
{
    return myName;
}

-(void) dealloc
{
    [myName release];
    [super dealloc];
}
@end
```

Delegate Servants in Objective-C

An alternate way to implement a servant is to use a delegate. `ICEObject` provides two constructors to do this:

Objective-C

```
@interface ICEObject NSObject <ICEObject, NSCopying>
// ...
-(id) initWithDelegate:(id)delegate;
+(id) objectWithDelegate:(id)delegate;
@end
```

The `delegate` parameter specifies an object to which the servant will delegate operation invocations. That object need not derive from the skeleton class; the only requirement on the delegate is that it must have an implementation of the methods corresponding to the Slice operations that are called by clients. As for derived servants, we suggest that the delegate adopt the skeleton protocol, so the compiler will emit a warning if you forget to implement one or more Slice operations in the delegate.

The implementation of the Slice operations in a delegate servant is exactly the same as for a derived servant.

Delegate servants are useful if you need to derive your servant implementation from a base class in order to access some functionality. In that case, you cannot also derive the servant from the generated skeleton class. A delegate servant gets around Objective-C's single inheritance limitation and saves you having to write a servant class that forwards each operation invocation to the delegate.

Another use case are different interfaces that share their implementation. As an example, consider the following Slice definitions:

Slice

```
interface Intf1 {
    void op1();
};

interface Intf2 {
    void op2();
};
```

If `op1` and `op2` are substantially similar in their implementation and share common state, it can be convenient to implement the servants for `Intf1` and `Intf2` using a common delegate class:

Objective-C

```
@interface Intf1AndIntf2 : NSObject<EXIntf1, EXIntf2>
    +(id) intf1AndIntf2;
@end

@implementation Intf1AndIntf2
    +(id) intf1AndIntf2 { /*...*/ }
    -(void) op1:(ICECurrent*)current { /*...*/ }
    -(void) op2:(ICECurrent*)current { /*...*/ }
@end
```

See Instantiating an Objective-C Servant for an example of how to instantiate delegate servants.

Delegate servants do not permit you to override operations that are inherited from `ICEObject` (such as `ice_ping`). Therefore, if you want to override `ice_ping`, for example, to implement a default servant, you must use a derived servant.

See Also

- Slice for a Simple File System
- Objective-C Mapping for Interfaces
- Parameter Passing in Objective-C
- Object Incarnation in Objective-C
- The Current Object
- Default Servants