Server-Side Python Mapping for Interfaces

Previous Next

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 Python

On the client side, interfaces map to proxy classes. On the server side, interfaces map to *skeleton classes*. A skeleton is an abstract base class from which you derive your servant class and define a 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 Python mapping generates the following definition for this interface:

Python

```
class Node(Ice.Object):
    def __init__(self):
        # ...

    #
    # Operation signatures.
    #
    # def name(self, current=None):
```

The important points to note here are:

- As for the client side, Slice modules are mapped to Python modules with the same name, so the skeleton class definitions are part of the `Filesystem` module.
- The name of the skeleton class is the same as the Slice interface (`Node`).
- The skeleton class contains a comment summarizing the method signature of each operation in the Slice interface.
- The skeleton class is an abstract base class because its constructor prevents direct instantiation of the class.
- The skeleton class inherits from `Ice.Object` (which forms the root of the Ice object hierarchy).

Back to Top ^

`Ice.Object` Base Class for Python Servants

`Object` is mapped to the `Ice.Object` class in Python:

Python

```
class Object(object):
    def ice_isA(self, id, current=None):
        # ...
    def ice_ping(self, current=None):
        # ...
    def ice_ids(self, current=None):
        # ...
    def ice_id(self, current=None):
        # ...

    @staticmethod
    def ice_staticId():
        # ...
```

The methods of `Ice.Object` behave as follows:

- `ice_isA`
  This method returns `true` if target object implements the given type ID, and `false` otherwise.
- `ice_ping`
  `ice_ping` provides a basic reachability test for the servant.
- `ice_ids`
  This method returns a string array representing all of the type IDs implemented by this servant, including `::Ice::Object`.
- `ice_id`
  This method returns the type ID of the most-derived interface implemented by this servant.
- `ice_staticID`
  This static method returns the type ID of the target class: `::Ice::Object` when called on `Ice.Object`.

Back to Top ^

Servant Classes in Python

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:

Python

```
import Filesystem

class NodeI(Filesystem.Node):
    def __init__(self, name):
        self._name = name

    def name(self, current=None):
        return self._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 `Filesystem._NodeDisp`, that is, it derives from its skeleton class.

As far as Ice is concerned, the `NodeI` class must implement only a single method: the `name` method that is defined in the `Node` interface. This makes the servant class a concrete class that can be instantiated. You can add other member functions 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 `_name` member and the `name` function returns its value.)

Back to Top ^

Server-Side Normal and `idempotent` Operations in Python

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 mapping for this interface is shown below:

Python

```
class Example(Ice.Object):
    # ...

    #
    # Operation signatures.
    #
    # def normalOp(self, current=None):
    # def idempotentOp(self, current=None):
```

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

Back to Top ^

See Also

- Slice for a Simple File System
- Python Mapping for Interfaces
- Parameter Passing in Python
- Raising Exceptions in Python

Previous

Next