Using the admin Object

A program can obtain a proxy for its `admin` object by calling the `getAdmin` operation on a communicator:

Slice

```
module Ice {
local interface Communicator {
    // ...
    Object* getAdmin();
};
};
```

This operation returns a null proxy if the administrative facility is disabled. The proxy returned by `getAdmin` cannot be used for invoking operations because it refers to the default facet and, as we mentioned previously, the `admin` object does not support a default facet. A program must first obtain a new version of the proxy that is configured with the name of a particular administrative facet before invoking operations on it. Although it cannot be used for invocations, the original proxy is still useful because it contains the endpoints of the administrative object adapter (`Ice.Admin`) and therefore the program may elect to export that proxy to a remote client.

To administer a program remotely, somehow you must obtain a proxy for the program's `admin` object. There are several ways for the administrative client to accomplish this:

- Construct the proxy itself, assuming that it knows the `admin` object's identity, facets, and endpoints. The format of the stringified proxy is as follows:
  `instance-name``/admin -f` *`admin-facet`*`:`*`admin-endpoints`*
  The identity category, represented here by *`instance-name`*, is the value of the `Ice.Admin.InstanceName` property or a UUID if that property is not defined. (Clearly, the use of a UUID makes the proxy much more difficult for a client to construct on its own.) The name of the administrative facet is supplied as the value of the `-f` option, and the endpoints of the `Ice.Admin` adapter appear last in the proxy.
- Invoke an application-specific interface for retrieving the `admin` object's proxy.
- Use the `getServerAdmin` operation on the `IceGrid::Admin` interface, if the remote program was activated by IceGrid (see IceGrid Server Activation).

Having obtained the proxy, the administrative client must select a facet before invoking any operations. For example, the code below shows how to obtain the configuration properties of the remote program:

C++

```
// C++
Ice::ObjectPrx adminObj = ...;
Ice::PropertiesAdminPrx propAdmin = Ice::PropertiesAdminPrx::checkedCast(adminObj, "Properties");
Ice::PropertyDict props = propAdmin->getPropertiesForPrefix("");
```

Here we used an overloaded version of `checkedCast` to supply the facet name of interest (`Properties`). We could have selected the facet using the proxy method `ice_facet` instead, as shown below:

C++

```
// C++
Ice::ObjectPrx adminObj = ...;
Ice::PropertiesAdminPrx propAdmin = Ice::PropertiesAdminPrx::checkedCast(
    adminObj->ice_facet("Properties"));
Ice::PropertyDict props = propAdmin->getPropertiesForPrefix("");
```

This code is functionally equivalent to the first example.

A remote client must also know (or be able to determine) which facets are available in the target server. Typically this information is statically configured in the client, since the client must also know the interface types of any facets that it uses. If an invocation on a facet raises `FacetNotExistException`, the client may have used an incorrect facet name, or the server may have disabled the facet in question.

See Also

- The admin Object
- The Administrative Object Adapter