# Well-Known Objects

On this page:



## Overview of Well-Known Objects

There are two types of indirect proxies: one specifies an identity and an object adapter identifier, while the other contains only an identity. The latter type of indirect proxy is known as a *well-known proxy*. A well-known proxy refers to a well-known object, that is, its identity alone is sufficient to allow the client to locate it. Ice requires all object identities in an application to be unique, but typically only a select few objects are able to be located only by their identities.

In earlier sections we showed the relationship between indirect proxies containing an object adapter identifier and the IceGrid configuration. Briefly, in order for a client to use a proxy such as `factory@EncoderAdapter`, an object adapter must be given the identifier `EncoderAdapter`.

A similar requirement exists for well-known objects. The registry maintains a table of these objects, which can be populated in a number of ways:

- statically in descriptors,
- programmatically using IceGrid's administrative interface,
- dynamically using an IceGrid administration tool.

The registry's database maps an object identity to a proxy. A locate request containing only an identity prompts the registry to consult this database. If a match is found, the registry examines the associated proxy to determine if additional work is necessary. For example, consider the well-known objects in the following table.

| Identity | Proxy |
|---|---|
| `Object1` | `Object1:tcp -p 10001` |
| `Object2` | `Object2@TheAdapter` |
| `Object3` | `Object3` |

The proxy associated with `Object1` already contains endpoints, so the registry can simply return this proxy to the client.

For `Object2`, the registry notices the adapter ID and checks to see whether it knows about an adapter identified as `TheAdapter`. If it does, it attempts to obtain the endpoints of that adapter, which may cause its server to be started. If the registry is successfully able to determine the adapter's endpoints, it returns a direct proxy containing those endpoints to the client. If the registry does not recognize `TheAdapter` or cannot obtain its endpoints, it returns the indirect proxy `Object2@TheAdapter` to the client. Upon receipt of another indirect proxy, the Ice run time in the client will try once more to resolve the proxy, but generally this will not succeed and the Ice run time in the client will raise a `NoEndpointException` as a result.

Finally, `Object3` represents a hopeless situation: how can the registry resolve `Object3` when its associated proxy refers to itself? In this case, the registry returns the proxy `Object3` to the client, which causes the client to raise `NoEndpointException`. Clearly, you should avoid this situation.

## Well-Known Object Types

The registry's database not only associates an identity with a proxy, but also a type. Technically, the "type" is an arbitrary string but, by convention, that string represents the most-derived Slice type of the object. For example, the Slice type ID of the encoder factory in our ripper application is `::Ripper::MP3EncoderFactory`.

Object types are useful when performing queries.

## Deploying Well-Known Objects

The object descriptor adds a well-known object to the registry. It must appear within the context of an adapter descriptor, as shown in the XML example below:

**XML**

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer" exe="/opt/ripper/bin/server" activation="on-demand">
                <adapter name="EncoderAdapter" id="EncoderAdapter" endpoints="tcp">
                    <object identity="EncoderFactory" type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
    </application>
</icegrid>
```

During deployment, the registry associates the identity `EncoderFactory` with the indirect proxy `EncoderFactory@EncoderAdapter`. If the adapter descriptor had omitted the adapter ID, the registry would have generated a unique identifier by combining the server ID and the adapter name.

In this example, the object's type is specified explicitly.

# Adding Well-Known Objects Programmatically

The `IceGrid::Admin` interface defines several operations that manipulate the registry's database of well-known objects:

**Slice**

```
module IceGrid {
interface Admin {
    ...
    void addObject(Object* obj)
        throws ObjectExistsException,
               DeploymentException;
    void updateObject(Object* obj)
        throws ObjectNotRegisteredException,
               DeploymentException;
    void addObjectWithType(Object* obj, string type)
        throws ObjectExistsException,
               DeploymentException;
    void removeObject(Ice::Identity id)
        throws ObjectNotRegisteredException,
               DeploymentException;
    ...
};
};
```

- `addObject`
  The `addObject` operation adds a new object to the database. The proxy argument supplies the identity of the well-known object. If an object with the same identity has already been registered, the operation raises `ObjectExistsException`. Since this operation does not accept an argument supplying the object's type, the registry invokes `ice_id` on the given proxy to determine its most-derived type. The implication here is that the object must be available in order for the registry to obtain its type. If the object is not available, `addObject` raises `DeploymentException`.

- `updateObject`
  The `updateObject` operation supplies a new proxy for the well-known object whose identity is encapsulated by the proxy. If no object with the given identity is registered, the operation raises `ObjectNotRegisteredException`. The object's type is not modified by this operation.

- `addObjectWithType`
  The `addObjectWithType` operation behaves like `addObject`, except the object's type is specified explicitly and therefore the registry does not attempt to invoke `ice_id` on the given proxy (even if the type is an empty string).

- `removeObject`
  The `removeObject` operation removes the well-known object with the given identity from the database. If no object with the given identity is registered, the operation raises `ObjectNotRegisteredException`.

The following C++ example produces the same result as the descriptor we deployed earlier:

**C++**

```
Ice::ObjectAdapterPtr adapter = communicator->createObjectAdapter("EncoderAdapter");
Ice::Identity ident = communicator->stringToIdentity("EncoderFactory");
FactoryPtr f= new FactoryI;
Ice::ObjectPrx factory = adapter->add(f, ident);
IceGrid::AdminPrx admin = // ...
try {
    admin->addObject(factory); // OOPS!
} catch (const IceGrid::ObjectExistsException &) {
    admin->updateObject(factory);
}
```

After obtaining a proxy for the `IceGrid::Admin` interface, the code invokes `addObject`. Notice that the code traps `ObjectExistsException` and calls `updateObject` instead when the object is already registered.

There is one subtle problem in this code: calling `addObject` causes the registry to invoke `ice_id` on our factory object, but we have not yet activated the object adapter. As a result, our program will hang indefinitely at the call to `addObject`. One solution is to activate the adapter prior to the invocation of `addObject`; another solution is to use `addObjectWithType` as shown below:

**C++**

```
Ice::ObjectAdapterPtr adapter = communicator->createObjectAdapter("EncoderAdapter");
Ice::Identity ident = communicator->stringToIdentity("EncoderFactory");
FactoryPtr f = new FactoryI;
Ice::ObjectPrx factory = adapter->add(f, ident);
IceGrid::AdminPrx admin = // ...
try {
    admin->addObjectWithType(factory, factory->ice_id());
} catch (const IceGrid::ObjectExistsException &) {
    admin->updateObject(factory);
}
```

# Adding Well-Known Objects with `icegridadmin`

The `icegridadmin` utility provides commands that are the functional equivalents of the Slice operations for managing well-known objects. We can use the utility to manually register the `EncoderFactory` object from our descriptors:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> object add "EncoderFactory@EncoderAdapter"
```

Use the `object list` command to verify that the object was registered successfully:

```
>>> object list
EncoderFactory
IceGrid/Query
IceGrid/Locator
IceGrid/Registry
IceGrid/InternalRegistry-Master
```

To specify the object's type explicitly, append it to the `object add` command:

```
>>> object add "EncoderFactory@EncoderAdapter" "::Ripper::MP3EncoderFactory"
```

Finally, the object is removed from the registry like this:

```
>>> object remove "EncoderFactory"
```

# Querying Well-Known Objects

The registry's database of well-known objects is not used solely for resolving indirect proxies. The database can also be queried interactively to find objects in a variety of ways. The `IceGrid::Query` interface supplies this functionality:

**Slice**

```
module IceGrid {
enum LoadSample {
    LoadSample1,
    LoadSample5,
    LoadSample15
};

interface Query {
    idempotent Object* findObjectById(Ice::Identity id);
    idempotent Object* findObjectByType(string type);
    idempotent Object* findObjectByTypeOnLeastLoadedNode(string type, LoadSample sample);
    idempotent Ice::ObjectProxySeq findAllObjectsByType(string type);
    idempotent Ice::ObjectProxySeq findAllReplicas(Object* proxy);
};
};
```

- `findObjectById`
  The `findObjectById` operation returns the proxy associated with the given identity of a well-known object. It returns a null proxy if no match was found.
- `findObjectByType`
  The `findObjectByType` operation returns a proxy for an object registered with the given type. If more than one object has the same type, the registry selects one at random. The operation returns a null proxy if no match was found.
- `findObjectByTypeOnLeastLoadedNode`
  The `findObjectByTypeOnLeastLoadedNode` operation considers the system load when selecting one of the objects with the given type. If the registry is unable to determine which node hosts an object (for example, because the object was registered with a direct proxy and not an adapter ID), the object is considered to have a load value of `1` for the purposes of this operation. The sample argument determines the interval over which the loads are averaged (one, five, or fifteen minutes). The operation returns a null proxy if no match was found.
- `findAllObjectsByType`
  The `findAllObjectsByType` operation returns a sequence of proxies representing the well-known objects having the given type. The operation returns an empty sequence if no match was found.
- `findAllReplicas`
  Given an indirect proxy for a replicated object, the `findAllReplicas` operation returns a sequence of proxies representing the individual replicas. An application can use this operation when it is necessary to communicate directly with one or more replicas.

Be aware that the operations accepting a `type` parameter are not equivalent to invoking `ice_isA` on each object to determine whether it supports the given type, a technique that would not scale well for a large number of registered objects. Rather, the operations simply compare the given type to the object's registered type or, if the object was registered without a type, to the object's most-derived Slice type as determined by the registry.

# Using Well-Known Objects in the Ripper Application

Well-known objects are another IceGrid feature we can incorporate into our ripper application.

## Adding Well-Known Objects to the Ripper Deployment

First we'll modify the descriptors to add two well-known objects:

**XML**

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer1" exe="/opt/ripper/bin/server" activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp">
                    <object identity="EncoderFactory1" type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
        <node name="Node2">
            <server id="EncoderServer2" exe="/opt/ripper/bin/server" activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp">
                    <object identity="EncoderFactory2" type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
    </application>
</icegrid>
```

At first glance, the addition of the well-known objects does not appear to simplify our client very much. Rather than selecting which of the two adapters receives the next task, we now need to select one of the well-known objects.

## Querying Ripper Objects with `findAllObjectsByType`

The `IceGrid::Query` interface provides a way to eliminate the client's dependency on object adapter identifiers and object identities. Since our factories are registered with the same type, we can search for all objects of that type:

**C++**

```
Ice::ObjectPrx proxy = communicator->stringToProxy("IceGrid/Query");
IceGrid::QueryPrx query = IceGrid::QueryPrx::checkedCast(proxy);
Ice::ObjectProxySeq seq;
string type = Ripper::MP3EncoderFactory::ice_staticId();
seq = query->findAllObjectsByType(type);
if (seq.empty()) {
    // no match
}
Ice::ObjectProxySeq::size_type index = ... // random number
Ripper::MP3EncoderFactoryPrx factory = Ripper::MP3EncoderFactoryPrx::checkedCast(seq[index]);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

This example invokes `findAllObjectsByType` and then randomly selects an element of the sequence.

## Querying Ripper Objects with `findObjectByType`

We can simplify the client further using `findObjectByType` instead, which performs the randomization for us:

**C++**

```
Ice::ObjectPrx proxy = communicator->stringToProxy("IceGrid/Query");
IceGrid::QueryPrx query = IceGrid::QueryPrx::checkedCast(proxy);
Ice::ObjectPrx obj;
string type = Ripper::MP3EncoderFactory::ice_staticId();
obj = query->findObjectByType(type);
if (!obj) {
    // no match
}
Ripper::MP3EncoderFactoryPrx factory = Ripper::MP3EncoderFactoryPrx::checkedCast(obj);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

## Querying Ripper Objects with `findObjectByTypeOnLeastLoadedNode`

So far the use of `IceGrid::Query` has allowed us to simplify our client, but we have not gained any functionality. If we replace the call to `findObjectByType` with `findObjectByTypeOnLeastLoadedNode`, we can improve the client by distributing the encoding tasks more intelligently. The change to the client's code is trivial:

**C++**

```
Ice::ObjectPrx proxy = communicator->stringToProxy("IceGrid/Query");
IceGrid::QueryPrx query = IceGrid::QueryPrx::checkedCast(proxy);
Ice::ObjectPrx obj;
string type = Ripper::MP3EncoderFactory::ice_staticId();
obj = query->findObjectByTypeOnLeastLoadedNode(type,
    IceGrid::LoadSample1);
if (!obj) {
    // no match
}
Ripper::MP3EncoderFactoryPrx factory = Ripper::MP3EncoderFactoryPrx::checkedCast(obj);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

## Ripper Progress Review

Incorporating intelligent load distribution is a worthwhile enhancement and is a capability that would be time consuming to implement ourselves. However, our current design uses only well-known objects in order to make queries possible. We do not really need the encoder factory object on each compute server to be individually addressable as a well-known object, a fact that seems clear when we examine the identities we assigned to them: `EncoderFactory1`, `EncoderFactory2`, and so on. IceGrid's replication features give us the tools we need to improve our design.

### See Also

- Terminology
- Type IDs
- Object Descriptor Element
- IceGrid Administrative Sessions
- IceGrid Administrative Utilities
- Object Adapter Replication