Using Identity Categories with Servant Locators

Our simple example always instantiates a servant of type `PhoneEntryI`. In other words, the servant locator implicitly is aware of the type of servant the incoming request is for. This is not a very realistic assumption for most servers because, usually, a server provides access to objects with several different interfaces. This poses a problem for our `locate` implementation: somehow, we need to decide inside `locate` what type of servant to instantiate. You have several options for solving this problem:

- Use a separate object adapter for each interface type and use a separate servant locator for each object adapter.

  This technique works fine, but has the down-side that each object adapter requires a separate transport endpoint, which is wasteful.

- Mangle a type identifier into the `name` component of the object identity.

  This technique uses part of the object identity to denote what type of object to instantiate. For example, in our file system application, we have directory and file objects. By convention, we could prepend a 'd' to the identity of every directory and prepend an '`f`' to the identity of every file. The servant locator then can use the first letter of the identity to decide what type of servant to instantiate:

  C++11
  C++98

```
std::shared_ptr<Ice::Object>
MyServantLocator::locate(const Ice::Current& current, std::shared_ptr<void>& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    auto name = c.id.name;
    auto realId = c.id.name.substr(1);
    try
    {
        if(name[0] == 'd')
        {
            // The request is for a directory.
            //
            DirectoryDetails d = DB_lookup(realId);
            return std::make_shared<DirectoryI>(d);
        }
        else
        {
            // The request is for a file.
            //
            FileDetails d = DB_lookup(realId);
            return std::make_shared<FileI>(d);
        }
    }
    catch(DatabaseNotFoundException&)
    {
        return 0;
    }
}
```

  C++98

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current& current, Ice::LocalObjectPtr& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;
    std::string realId = c.id.name.substr(1);
    try
    {
        if(name[0] == 'd')
        {
            // The request is for a directory.
            //
            DirectoryDetails d = DB_lookup(realId);
            return new DirectoryI(d);
        }
        else
        {
            // The request is for a file.
            //
            FileDetails d = DB_lookup(realId);
            return new FileI(d);
        }
    }
    catch(DatabaseNotFoundException&)
    {
        return 0;
    }
}
```

While this works, it is awkward: not only do we need to parse the `name` member to work out what type of object to instantiate, but we also need to modify the implementation of `locate` whenever we add a new type to our application.

- Use the `category` member of the object identity to denote the type of servant to instantiate.

  This is the recommended approach: for every interface type, we assign a separate identifier as the value of the `category` member of the object identity. (For example, we can use 'd' for directories and 'f' for files.) Instead of registering a single servant locator, we create two different servant locator implementations, one for directories and one for files, and then register each locator for the appropriate category:

  C++11

```
class DirectoryLocator : public Ice::ServantLocator
{
public:

    virtual std::shared_ptr<Ice::Object> locate(const Ice::Current& current, std::
shared_ptr<void>& cookie) override
    {
        // Code to locate and instantiate a directory here...
    }

    virtual void finished(const Ice::Current& current, const std::shared_ptr<Ice::Object>& servant,
const std::shared_ptr<void>& cookie) override
    {
    }

    virtual void deactivate(const std::string& category) override
    {
    }
};

class FileLocator : public Ice::ServantLocator
{
public:

    virtual std::shared_ptr<Ice::Object> locate(const Ice::Current& current, std::
shared_ptr<void>& cookie) override
    {
        // Code to locate and instantiate a file here...
    }

    virtual void finished(const Ice::Current& current, const std::shared_ptr<Ice::Object> servant,
const std::shared_ptr<void>& cookie) override
    {
    }

    virtual void deactivate(const std::string& category) override
    {
    }
};

// ...

// Register two locators, one for directories and
// one for files.
//
adapter->addServantLocator(std::make_shared<DirectoryLocator>(), "d");
adapter->addServantLocator(std::make_shared<FileLocator>(), "f");
```

C++98

```
class DirectoryLocator : public Ice::ServantLocator
{
public:

    virtual Ice::ObjectPtr locate(const Ice::Current& current, Ice::LocalObjectPtr& cookie)
    {
        // Code to locate and instantiate a directory here...
    }

    virtual void finished(const Ice::Current& current, const Ice::ObjectPtr& servant, const Ice::
LocalObjectPtr& cookie)
    {
    }

    virtual void deactivate(const std::string& category)
    {
    }
};

class FileLocator : public Ice::ServantLocator
{
public:

    virtual Ice::ObjectPtr locate(const Ice::Current& current, Ice::LocalObjectPtr& cookie)
    {
        // Code to locate and instantiate a file here...
    }

    virtual void finished(const Ice::Current& current, const Ice::ObjectPtr& servant, const Ice::
LocalObjectPtr& cookie)
    {
    }

    virtual void deactivate(const std::string& category)
    {
    }
};

// ...

// Register two locators, one for directories and
// one for files.
//
adapter->addServantLocator(new DirectoryLocator(), "d");
adapter->addServantLocator(new FileLocator(), "f");
```

- Yet another option is to use the `category` member of the object identity, but to use a single default servant locator (that is, a locator for the empty category). With this approach, all invocations go to the single default servant locator, and you can switch on the `category` value inside the implementation of the `locate` operation to determine which type of servant to instantiate. However, this approach is harder to maintain than the previous one; the `category` member of the Ice object identity exists specifically to support servant locators, so you might as well use it as intended.

Back to Top ^

See Also

- Servant Locator Example
- Object Identity

Previous Next