Registering a Servant Locator

Previous Next

On this page:

Servant Locator Registration

An object adapter does not automatically know when you create a servant locator. Instead, you must explicitly register a servant locator with the object adapter:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        // ...

        void addServantLocator(ServantLocator locator, string category);

        ServantLocator removeServantLocator(string category);

        ServantLocator findServantLocator(string category);

        // ...
    }
}
```

As you can see, the object adapter allows you to add, remove, and find servant locators. Note that, when you register a servant locator, you must provide an argument for the `category` parameter. The value of the `category` parameter controls which object identities the servant locator is responsible for: only object identities with a matching `category` member trigger a corresponding call to `locate`. An incoming request for which no explicit entry exists in the active servant map (ASM) and with a category for which no servant locator is registered returns an `ObjectNotExistException` to the client.

`addServantLocator` has the following semantics:

- You can register exactly one servant locator for a specific category. Attempts to call `addServantLocator` for the same category more than once raise an `AlreadyRegisteredException`.
- You can register different servant locators for different categories, or you can register the same single servant locator multiple times (each time for a different category). In the former case, the category is implicit in the servant locator instance that is called by the Ice run time; in the latter case, the implementation of `locate` can find out which category the incoming request is for by examining the object identity member of the `Current` object that is passed to `locate`.
- It is legal to register a servant locator for the empty category. Such a servant locator is known as a *default servant locator*: if a request comes in for which no entry exists in the ASM, and whose category does not match the category of any other registered servant locator, the Ice run time calls `locate` on the default servant locator.

`removeServantLocator` removes and returns the servant locator for a specific category (including the empty category) with the following semantics:

- If no servant locator is registered for the specified category, the operation raises `NotRegisteredException`.
- Once a servant locator is successfully removed for the specified category, the Ice run time guarantees that no new incoming requests for that category are dispatched to the servant locator.
- A call to `removeServantLocator` returns immediately without waiting for the completion of any pending requests on that servant locator; such requests still complete normally by calling `finished` on the servant locator.
- Removing a servant locator does not cause Ice to invoke `deactivate` on that servant locator, as `deactivate` is only called when a registered servant locator's object adapter is destroyed.

`findServantLocator` allows you to retrieve the servant locator for a specific category (including the empty category). If no match is found, the operation returns null.

Back to Top ^

Call Dispatch Semantics for Servant Locators

The preceding rules may seem complicated, so here is a summary of the actions taken by the Ice run time to locate a servant for an incoming request.

Every incoming request implicitly identifies a specific object adapter for the request (because the request arrives at a specific transport endpoint and, therefore, identifies a particular object adapter). The incoming request carries an object identity that must be mapped to a servant. To locate a servant, the Ice run time goes through the following steps, in the order shown:

1. Look for the identity in the ASM. If the ASM contains an entry, dispatch the request to the corresponding servant.
2. If the category of the incoming object identity is non-empty, look for a default servant that is registered for that category. If such a default servant is registered, dispatch the request to that servant.
3. If the category of the incoming object identity is empty, or no default servant could be found for the category in step 2, look for a default servant that is registered for the empty category. If such a default servant is registered, dispatch the request to that servant.
4. If the category of the incoming object identity is non-empty and no servant could be found in the preceding steps, look for a servant locator that is registered for that category. If such a servant locator is registered, call `locate` on the servant locator and, if `locate` returns a servant, dispatch the request to that servant, followed by a call to `finished`; otherwise, if the call to `locate` returns null, raise `ObjectNotExistException` or F`acetNotExistException` in the client.
5. If the category of the incoming object identity is empty, or no servant locator could be found for the category in step 4, look for a default servant locator (that is, a servant locator that is registered for the empty category). If a default servant locator is registered, dispatch the request as for step 4.
6. Raise `ObjectNotExistException` or `FacetNotExistException` in the client. (`ObjectNotExistException` is raised if the ASM does not contain a servant with the given identity at all, `FacetNotExistException` is raised if the ASM contains a servant with a matching identity, but a non-matching facet.)

It is important to keep these call dispatch semantics in mind because they enable a number of powerful implementation techniques. Each technique allows you to streamline your server implementation and to precisely control the trade-off between performance, memory consumption, and scalability. To illustrate the possibilities, we will outline a number of the most common implementation techniques.

Back to Top ^

See Also

- Object Adapters
- Object Identity
- The Active Servant Map
- Default Servants
- Versioning
- Servant Locator Example

Previous Next