Default Servants

Previous Next

On this page:

Overview of Default Servants

The Active Servant Map (ASM) is a simple lookup table that maintains a one-to-one mapping between object identities and servants. Although the ASM is easy to understand and offers efficient indexing, it does not scale well when the number of objects is very large. Scalability is a common problem with object-oriented middleware: servers frequently are used as front ends to large databases that are accessed remotely by clients. The server's job is to present an object-oriented view to clients of a very large number of records in the database. Typically, the number of records is far too large to instantiate servants for even a fraction of the database records.

A common technique for solving this problem is to use *default servants*. A default servant is a servant that, for each request, takes on the persona of a different Ice object. In other words, the servant changes its behavior according to the object identity that is accessed by a request, on a per-request basis. In this way, it is possible to allow clients access to an unlimited number of Ice objects with only a single servant in memory. A default servant is essentially a specialized version of a servant locator that satisfies the majority of use cases with a simpler API, whereas a servant locator provides more flexibility for those applications that require it.

Default servant implementations are attractive not only because of the memory savings they offer, but also because of the simplicity of implementation: in essence, a default servant is a facade [1] to the persistent state of an object in the database. This means that the programming required to implement a default servant is typically minimal: it simply consists of the code required to read and write the corresponding database records.

A default servant is a regular servant that you implement and register with an object adapter. For each incoming request, the object adapter first attempts to locate a servant in its ASM. If no servant is found, the object adapter dispatches the request to a default servant. With this design, a default servant is the object adapter's servant of last resort if no match was found in the ASM.

Implementing a default servant requires a somewhat different mindset than the typical "one servant per Ice object" strategy used in less advanced applications. The most important quality of a default servant is its statelessness: it must be prepared to dispatch multiple requests simultaneously for different objects. The price we have to pay for the unlimited scalability and reduced memory footprint is performance: default servants typically make a database access for every invoked operation, which is obviously slower than caching state in memory as part of a servant that has been added to the ASM. However, this does not mean that default servants carry an unacceptable performance penalty: databases often provide sophisticated caching, so even though the operation implementations read and write the database, as long as they access cached state, performance may be entirely acceptable.

Back to Top ^

Default Servant API

The default servant API consists of the following operations in the object adapter interface:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        void addDefaultServant(Object servant, string category);
        Object removeDefaultServant(string category);
        Object findDefaultServant(string category);

        // ...
    }
}
```

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

`addDefaultServant` has the following semantics:

- You can register exactly one default servant for a specific category. Attempts to call `addDefaultServant` for the same category more than once raise an `AlreadyRegisteredException`.
- You can register different default servants for different categories, or you can register the same single default servant multiple times (each time for a different category). In the former case, the category is implicit in the default servant instance that is called by the Ice run time; in the latter case, the servant can find out which category the incoming request is for by examining the object identity member of the `Current` object that is passed to the dispatched operation.
- It is legal to register a default servant for the empty category. Such a servant is used 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 default servant.

`removeDefaultServant` removes the default servant for the specified category. Attempts to remove a non-existent default servant raise `NotRegisteredException`. The operation returns the removed default servant. Once a default servant 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.

The `findDefaultServant` operation allows you to retrieve the default servant for a specific category (including the empty category). If no default servant is registered for the specified category, `findDefaultServant` returns null.

Back to Top ^

Threading Guarantees for Default Servants

The threading semantics for a default servant are no different than for a servant registered in the ASM: operations may be dispatched on a default servant concurrently, for the same object identity or for different object identities. If you have configured the communicator with multiple dispatch threads, your default servant must protect access to shared data with appropriate locks.

Back to Top ^

Call Dispatch Semantics for Default Servants

This section summarizes 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 `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.)
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.

Back to Top ^

Guidelines for Implementing Default Servants

This section provides some guidelines to assist you in implementing default servants effectively.

Object Identity is the Key

When an incoming request is dispatched to the default servant, the target object identity is provided in the `Current` argument. The `name` field of the identity typically supplies everything the default servant requires in order to satisfy the request. For instance, it may serve as the key in a database query, or even hold an encoded structure in some proprietary format that your application uses to convey more than just a string.

Naturally, the client can also pass arguments to the operation that assist the default servant in retrieving whatever state it requires. However, this approach can easily introduce implementation artifacts into your Slice interfaces, and in most cases the client should not need to know that the server is implemented with a default servant. If at all possible, use only the object identity.

Back to Top ^

Minimize Contention

For better scalability, the default servant's implementation should strive to eliminate contention among the dispatch threads. As an example, when a database holds the default servant's state, each of the servant's operations usually begins with a query. Assuming that the database API is thread-safe, the servant needs to perform no explicit locking of its own. With a copy of the state in hand, the implementation can work with function-local data to satisfy the request.

Back to Top ^

Combine Strategies

The ASM still plays a useful role even in applications that are ideally suited for default servants. For example, there is no need to implement a singleton object as a default servant: if there can only be one instance of the object, implementing it as a default servant does nothing to improve your application's scalability.

Applications often install a handful of servants in the ASM while servicing the majority of requests in a default servant. For example, a database application might install a singleton query object in the ASM while using a default servant to process all invocations on the database records.

Back to Top ^

Categories Denote Interfaces

In general, all of the objects serviced by a default servant must have the same interface. If you only need a default servant for one interface, you can register the default servant with an empty category string. However, to implement several interfaces, you will need a default servant implementation for each one. Furthermore, you must take steps to ensure that the object adapter dispatches an incoming request to the appropriate default servant. The `category` field of the object identity is intended to serve this purpose.

For example, a process control system might have interfaces named `Sensor` and `Switch`. To direct requests to the proper default servant, the application uses the symbol `Sensor` or `Switch` as the category of each object's identity, and registers corresponding default servants having those same categories with the object adapter.

Back to Top ^

Plan for the Future

If you suspect that you might eventually need to implement more than one interface with default servants, we recommend using a non-empty category even if you start out having only one default servant. Adding another default servant later becomes much easier if the application is already designed to operate correctly with categories.

Back to Top ^

Throw exceptions

If a request arrives for an object that no longer exists, it is the default servant's responsibility to raise `ObjectNotExistException` to properly manage object life cycles.

Back to Top ^

Handle `ice_ping`

One issue you need to be aware of with default servants is the need to override `ice_ping`: the default implementation of `ice_ping` that the servant inherits from its skeleton class always succeeds. For servants that are registered with the ASM, this is exactly what we want; however, for default servants, `ice_ping` must fail if a client uses a proxy to an Ice object that no longer exists. To avoid getting successful `ice_ping` invocations for non-existent Ice objects, you must override `ice_ping` in the default servant. The implementation must check whether the object identity for the request denotes a still-existing Ice object and, if not, raise `ObjectNotExistException`.

It is good practice to override `ice_ping` if you are using default servants. Because you cannot override operations on `Ice::Object` using a Java or C# tie servant (or an Objective-C delegate servant), you must implement default servants by deriving from the generated skeleton class if you choose to override `ice_ping`.

Back to Top ^

Consider Interceptors

A dispatch interceptor is often installed as a default servant.

Back to Top ^

Use a `Blobject` Default Servant to Forward Messages

Message forwarding services, such as Glacier2, can be implemented simply and efficiently with a `Blobject` default servant. Such a servant simply chooses a destination to forward a request to, without decoding any of the parameters.

Back to Top ^

See Also

- The Active Servant Map
- Object Identity
- Servant Locators
- Object Adapters
- The Current Object
- The Ice Threading Model
- Versioning
- Dispatch Interceptors
- Dynamic Invocation and Dispatch Overview
- Glacier2

References

1. Gamma, E., et al. 1994. Design Patterns. Reading, MA: Addison-Wesley.

Previous

Next