Dispatch Interceptors

A dispatch interceptor is a server-side mechanism that allows you to intercept incoming client requests before they are given to a servant. The interceptor can examine the incoming request; in particular, it can see whether the request dispatch is collocation-optimized and examine the `Current` information for the request.

A dispatch interceptor can dispatch a request to a servant and check whether the dispatch was successful; if not, the interceptor can choose to retry the dispatch. This functionality is useful to automatically retry requests that have failed due to a recoverable error condition, such as a database deadlock exception. (Freeze uses dispatch interceptors for this purpose in its evictor implementations.)

On this page:

Dispatch Interceptor API

Dispatch interceptors are not defined in Slice, but are provided as an API that is specific to each programming language. The remainder of this section presents the interceptor API for C++; for Java and .NET, the API is analogous, so we do not show it here.

In C++, a dispatch interceptor has the following interface:

C++

```
namespace Ice {
    class DispatchInterceptor : public virtual Object {
    public:
        virtual DispatchStatus dispatch(Request&) = 0;
    };

    typedef IceInternal::Handle<DispatchInterceptor> DispatchInterceptorPtr;
}
```

Note that a `DispatchInterceptor` *is-a* `Object`, that is, you use a dispatch interceptor as a servant.

To create a dispatch interceptor, you must derive a class from `DispatchInterceptor` and provide an implementation of the pure virtual `dispatch` function. The job of `dispatch` is to pass the request to the servant and to return a dispatch status, defined as follows:

C++

```
namespace Ice {
    enum DispatchStatus {
        DispatchOK, DispatchUserException, DispatchAsync
    };
}
```

The enumerators indicate how the request was dispatched:

- `DispatchOK`
  The request was dispatched synchronously and completed without an exception.
- `DispatchUserException`
  The request was dispatched synchronously and raised a user exception.
- `DispatchAsync`
  The request was dispatched successfully as an asynchronous request; the result of the request is not available to the interceptor because the result is delivered to the AMD callback when the request completes.

The Ice run time provides basic information about the request to the `dispatch` function in the form of a `Request` object:

C++

```
namespace Ice {
    class Request {
    public:
        virtual bool isCollocated();
        virtual const Current& getCurrent();
    };
}
```

- `isCollocated` returns true if the dispatch is directly into the target servant as a collocation-optimized dispatch. If the dispatch is not collocation-optimized, the function returns false.
- `getCurrent` provides access to the `Current` object for the request, which provides access to information about the request, such as the object identity of the target object, the object adapter used to dispatch the request, and the operation name.

Note that `Request`, for performance reasons, is *not* thread-safe. This means that you must not concurrently dispatch from different threads using the same `Request` object. (Concurrent dispatch for different requests does not cause any problems.)

To use a dispatch interceptor, you instantiate your derived class and register it as a servant with the Ice run time in the usual way, such as by adding the interceptor to the Active Servant Map (ASM), or returning the interceptor as a servant from a call to `locate` on a servant locator.

Objective-C Mapping for Dispatch Interceptors

The Objective-C mapping in Ice Touch does not support AMD, therefore the return type of the `dispatch` method is simplified to a boolean:

Objective-C

```
@protocol ICEDispatchInterceptor <ICEObject>
-(BOOL) dispatch:(id<ICERequest>)request;
@end
```

A return value of `YES` is equivalent to `DispatchOK` and indicates that the request completed without an exception. A return value of `NO` is equivalent to `DispatchUserException`.

Similarly, the `ICERequest` protocol omits the `isCollocated` method because collocation optimization is not supported.

Using a Dispatch Interceptor

Your implementation of the `dispatch` function must dispatch the request to the actual servant. Here is a very simple example implementation of an interceptor that dispatches the request to the servant passed to the interceptor's constructor:

C++

```
class InterceptorI : public Ice::DispatchInterceptor {
public:
    InterceptorI(const Ice::ObjectPtr& servant)
        : _servant(servant) {}

    virtual Ice::DispatchStatus dispatch(Ice::Request& request) {
        return _servant->ice_dispatch(request);
    }

    Ice::ObjectPtr _servant;
};
```

Note that our implementation of `dispatch` calls `ice_dispatch` on the target servant to dispatch the request. `ice_dispatch` does the work of actually (synchronously) invoking the operation.

Also note that `dispatch` returns whatever is returned by `ice_dispatch`. For synchronous dispatch, you should always implement your interceptor in this way and not change this return value.

We can use this interceptor to intercept requests to a servant of any type as follows:

C++

```
ExampleIPtr servant = new ExampleI;
Ice::DispatchInterceptorPtr interceptor = new InterceptorI(servant);
adapter->add(interceptor, communicator->stringToIdentity("ExampleServant"));
```

Note that, because dispatch interceptor *is-a* servant, this means that the servant to which the interceptor dispatches need not be the actual servant. Instead, it could be another dispatch interceptor that ends up dispatching to the real servant. In other words, you can chain dispatch interceptors; each interceptor's `dispatch` function is called until, eventually, the last interceptor in the chain dispatches to the actual servant.

A more interesting use of a dispatch interceptor is to retry a call if it fails due to a recoverable error condition. Here is an example that retries a request if it raises a local exception defined in Slice as follows:

Slice

```
local exception DeadlockException { /* ... */ };
```

Note that this is a `local` exception. Local exceptions that are thrown by the servant propagate to `dispatch` and can be caught there. A database might throw such an exception if the database detects a locking conflict during an update. We can retry the request in response to this exception using the following `dispatch` implementation:

C++

```
virtual Ice::DispatchStatus dispatch(Ice::Request& request) {
    while (true) {
        try {
            return _servant->ice_dispatch(request);
        } catch (const DeadlockException&) {
            // Happens occasionally
        }
    }
}
```

Of course, a more robust implementation might limit the number of retries and possibly add a delay before retrying.

You can also retry an asynchronous dispatch. In this case, each asynchronous dispatch attempt creates a new AMD callback object.

- If the response for the retried request has been sent already, the interceptor receives a `ResponseSentException`. Your interceptor must either not handle this exception (or rethrow it) or return `DispatchAsync`.
- If the response for the request has not been sent yet, the Ice run time ignores any call to `ice_response` or `ice_exception` on the old AMD callback.

If an operation throws a user exception (as opposed to a local exception), the user exception cannot be caught by `dispatch` as an exception but, instead, is reported by the return value of `ice_dispatch`: a return value of `DispatchUserException` indicates that the operation raised a user exception. You can retry a request in response to a user exception as follows:

C++

```
virtual Ice::DispatchStatus dispatch(Ice::Request& request) {
    Ice::DispatchStatus d;
    do {
        d = _servant->ice_dispatch(request);
    } while (d == Ice::DispatchUserException);
    return d;
}
```

This is fine as far as it goes, but not particularly useful because the preceding code retries if *any* kind of user exception is thrown. However, typically, we want to retry a request only if a *specific* user exception is thrown. The problem here is that the `dispatch` function does not have direct access to the actual exception that was thrown — all it knows is that *some* user exception was thrown, but not which one.

To retry a request for a specific user exception, you need to implement your servants such that they leave some "footprint" behind if they throw the exception of interest. This allows your request interceptor to test whether the user exception should trigger a retry. There are various techniques you can use to achieve this. For example, you can use thread-specific storage to test a retry flag that is set by the servant if it throws the exception or, if you use transactions, you can attach the retry flag to the transaction context. However, doing so is more complex; the intended use case is to permit retry of requests in response to local exceptions, so we suggest you retry requests only for local exceptions.

The most common use case for a dispatch interceptor is as a default servant. Rather than having an explicit interceptor for individual servants, you can register a dispatch interceptor as default servant. You can then choose the "real" servant to which to dispatch the request inside `dispatch`, prior to calling `ice_dispatch`. This allows you to intercept and selectively retry requests based on their outcome, which cannot be done using a servant locator.

See Also

- The Current Object
- Location Transparency
- The Active Servant Map
- Servant Locators
- Freeze Evictors
- Default Servants