# Design Considerations for Request Contexts

Previous
Next

On this page:

- Request Context Interactions
- Request Context Use Cases
- Recommendations for Request Contexts

## Request Context Interactions

If you use explicit, per-proxy, and implicit contexts, it is important to be aware of their interactions:

- If you send an explicit context with an invocation, *only* that context is sent with the call, regardless of whether the proxy has a per-proxy context and whether the communicator has an implicit context.
- If you send an invocation via a proxy that has a per-proxy context, and the communicator also has an implicit context, the contents of the per-proxy and implicit context dictionaries are combined, so the *combination* of context entries of both contexts is transmitted to the server. If the per-proxy context and the implicit context contain the same key, but with different values, the *per-proxy* value takes precedence.

Back to Top ^

## Request Context Use Cases

The purpose of `Ice::Context` is to permit services to be added to Ice that require some contextual information with every request. Contextual information can be used by services such as a transaction service (to provide the context of a currently established transaction) or a security service (to provide an authorization token to the server). IceStorm uses the context to provide an optional `cost` parameter to the service that influences how the service propagates messages to down-stream subscribers, and Glacier2 uses the context to influence request routing.

In general, services that require such contextual information can be implemented much more elegantly using contexts because this hides explicit Slice parameters that would otherwise have to be supplied by the application programmer with every call. For a request-forwarding intermediary service such as IceStorm or Glacier2, using a parameter is not an option because the service does not know the signatures of the requests that it is forwarding, but the service does have access to the context. For this use case, the context is a convenient way for the caller to annotate a request.

In addition, contexts, because they are optional, permit a single Slice definition to apply to implementations that use the context, as well as to implementations that do not use it. In this way, to add transactional semantics to an existing service, you do not need to modify the Slice definitions to add an extra parameter to each operation. (Adding an extra parameter would not only be inconvenient for clients, but would also split the type system into two halves: without contexts, we would need different Slice definitions for transactional and non-transactional implementations of (conceptually) a single service.)

Finally, per-proxy contexts permit context information to be passed through intermediate parts of your program without cooperation of those intermediate parts. For example, suppose you set a per-proxy context on a proxy and then pass that proxy to another function. When that function uses the proxy to invoke an operation, the per-proxy context will still be sent. In other words, per-proxy contexts allow you to transparently propagate information via intermediaries that are ignorant of the presence of any context.

Keep in mind though that this works only within a single process. If you stringify a proxy or transmit it as a parameter over the wire, the per-proxy context is *not* preserved. (Ice does not write the per-proxy context into stringified proxies and does not marshal the per-proxy context when a proxy is marshaled.)

Back to Top ^

## Recommendations for Request Contexts

Contexts are a powerful mechanism for transparent propagation of context information, *if used correctly*. In particular, you may be tempted to use contexts as a means of versioning an application as it evolves over time. For example, version 2 of your application may accept two parameters on an operation that, in version 1, used to accept only a single parameter. Using contexts, you could supply the second parameter as a name-value pair to the server and avoid changing the Slice definition of the operation in order to maintain backward compatibility.

We *strongly* urge you to resist any temptation to use contexts in this manner. The strategy is fraught with problems:

- Missing context
  There is nothing that would compel a client to actually send a context when the server expects to receive a context: if a client forgets to send a context, the server, somehow, has to make do without it (or throw an exception).

- Missing or incorrect keys
  Even if the client does send a context, there is no guarantee that it has set the correct key. (For example, a simple spelling error can cause the client to send a value with the wrong key.)

- Incorrect values
  The value of a context is a string, but the application data that is to be sent might be a number, or it might be something more complex, such as a

structure with several members. This forces you to encode the value into a string and decode the value again on the server side. Such parsing is tedious and error prone, and far less efficient than sending strongly-typed parameters. In addition, the server has to deal with string values that fail to decode correctly (for example, because of an encoding error made by the client).

None of the preceding problems can arise if you use proper Slice parameters: parameters cannot be accidentally omitted and they are strongly typed, making it much less likely for the client to accidentally send a meaningless value. Furthermore, you can use optional parameters to modify the signature of an operation without breaking backward compatibility.

Contexts are meant to be used to transmit simple tokens (such as a transaction identifier) for services that cannot be reasonably implemented without them; you should restrict your use of contexts to that purpose and resist any temptation to use contexts for any other purpose.

Finally, be aware that, if a request is routed via one or more Ice routers, contexts may be dropped by intermediate routers if they consider them illegal. This means that, in general, you cannot rely on an arbitrary context value that is created by an application to actually still be present when a request arrives at the server — only those context values that are known to routers and that are considered legitimate are passed on. It follows that you should not abuse contexts to pass things that really should be passed as parameters.

Back to Top ^

## See Also

- Explicit Request Contexts
- Per-Proxy Request Contexts
- Implicit Request Contexts
- Optional Values
- Versioning
- IceStorm

Previous

Next