Asynchronous APIs in JavaScript

Previous Next

Given JavaScript's lack of support for threads, Ice for JavaScript uses asynchronous semantics in every situation that has the potential to block, including both local and non-local invocations. Synchronous proxy invocations are not supported.

Here is an example of a simple proxy invocation:

JavaScript

```
let proxy = HelloPrx.uncheckedCast(...);
let promise = proxy.sayHello();
promise.then(
    () => {
        // handle success...
    },
    ex => {
        // handle failure...
    }
);
```

The API design is similar to that of other asynchronous Ice language mappings in that the return value of a proxy invocation is an instance of `Ice.AsyncResult`, through which the program configures callbacks to handle the eventual success or failure of the invocation. The JavaScript implementation of `Ice.AsyncResult` derives from `Ice.Promise` which is a thin extension of the standard JavaScript Promise.

Certain operations of local Ice run-time objects can also block, either because their implementations make remote invocations, or because their purpose is to block until some condition is satisfied:

- `Communicator::destroy`
- `Communicator::waitForShutdown`
- `Communicator::createObjectAdapterWithRouter`
- `Communicator::flushBatchRequests`
- `Connection::close`
- `Connection::flushBatchRequests`
- `ObjectPrx::ice_getConnection`

These operations use the same asynchronous API as for proxy invocations. The example below shows how to call `ice_getConnection` on a proxy:

JavaScript

```
let communicator = ...;
let proxy = communicator.stringToProxy("...");
proxy.ice_getConnection().then(
    connection => {
        // got connection...
    },
    ex => {
        // connection failed...
    }
);
```

Back to Top ^

Previous Next