JavaScript Mapping for Operations

Previous Next

JavaScript's event-driven APIs makes a synchronous invocation model impractical, therefore the JavaScript language mapping provides only an asynchronous model.

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests and invocations never block. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

On this page:

- Basic Asynchronous API in JavaScript
  - Asynchronous Proxy Methods in JavaScript
  - Passing Parameters in JavaScript
  - Null Parameters in JavaScript
  - Optional Parameters in JavaScript
  - Asynchronous Exception Semantics in JavaScript
- AsyncResult Type in JavaScript
- Promise Type in JavaScript
  - Basic Promise Concepts
  - Getting Started with Promises
  - Chaining Promises
  - Handling Errors with Promises
  - Passing Values between Promises
  - Chaining Asynchronous Events with Promises
  - Promise API
- Completion Callbacks in JavaScript
- Asynchronous Oneway Invocations in JavaScript
- Asynchronous Batch Requests in JavaScript

Basic Asynchronous API in JavaScript

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
        string getAddress(int number);
    }
}
```

Back to Top ^

Asynchronous Proxy Methods in JavaScript

`slice2js` generates the following asynchronous proxy methods:

JavaScript

```
EmployeesPrx = class extends Ice.ObjectPrx
{
    getName(number, context) { ... }
    getAddress(number, context) { ... }
}
```

As you can see, a Slice operation maps to a method of the same name. (Each function accepts an optional trailing per-invocation context.)

The `getName` function, for example, sends an invocation of `getName`. Because proxy invocations do not block, the application can do other things while the operation is in progress. The application receives notification about the completion of the request by registering callbacks for success and failure.

The `getName` function, as with all functions corresponding to Slice operations, returns a value of type `Ice.AsyncResult`. This value contains the state that the Ice run time requires to keep track of the asynchronous invocation.

A proxy function has one parameter for each in-parameter of the corresponding Slice operation. For example, consider the following operation:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `op` member function has the following signature:

JavaScript

```
function op(inp1, inp2, context); // Returns Ice.AsyncResult
```

The operation's return value, as well as the values of its two out-parameters, are delivered to the success callback upon completion of the request.

Back to Top ^

Passing Parameters in JavaScript

The parameter passing rules for the JavaScript mapping are very simple: parameters are passed either by value (for simple types) or by reference (for complex types). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following proxy methods for these definitions:

JavaScript

```
ClientToServerPrx = class extends Ice.ObjectPrx
{
    op1(i, f, b, s, context) { ... }
    op2(ns, ss, st, context) { ... }
    op3(proxy, context) { ... }
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

JavaScript

```
let p = ...;          // Get ClientToServerPrx proxy...

p.op1(42, 3.14, true, "Hello world!");  // Pass simple literals

let i = 42;
let f = 3.14;
let b = true;
let s = "Hello world!";
p.op1(i, f, b, s);                      // Pass simple variables

let ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
let ss = [];
ss.push("Hello world!");
let st = new StringTable();
st.set(0, ss);
p.op2(ns, ss, st);                      // Pass complex variables

p.op3(p);                               // Pass proxy
```

Back to Top ^

Null Parameters in JavaScript

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `null`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `null` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid a run-time error. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as `null` or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

Back to Top ^

Optional Parameters in JavaScript

Optional parameters use the same mapping as required parameters. The only difference is that `undefined` can be passed as the value of an optional parameter or return value to indicate an "unset" condition. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float value);
```

A client can invoke this operation as shown below:

JavaScript

```
proxy.execute("--file log.txt").then(
    r => {
                let [retval, value] = r;
        if(value !== undefined)
        {
            console.log("value =", value);
        }
    });
```

A well-behaved program must always compare an optional parameter to `undefined` prior to using its value.

For Slice types that support null semantics, such as proxies and objects by value, passing `null` for an optional parameter has a different meaning than passing `undefined`:

- `null` means a value was supplied for the parameter, and that value happens to be `null`
- `undefined` means no value was supplied for the parameter

`undefined` is not a legal value for required parameters.

Back to Top ^

Asynchronous Exception Semantics in JavaScript

If an invocation raises an exception, the exception is delivered to the failure callback, even if the actual error condition for the exception was encountered during the proxy function ("on the way out"). The advantage of this behavior is that all exception handling is located in the failure callback (instead of being present twice, once where the proxy function is called, and again in the failure callback).

There are two exceptions to this rule:

- if you destroy the communicator and then make an asynchronous invocation, the `opAsync` method throws `CommunicatorDestroyedException` directly.
- a call to an `Async` function can throw `TwowayOnlyException`. An `Async` function throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy.

Back to Top ^

`AsyncResult` Type in JavaScript

The `AsyncResult` object returned by an Ice API call encapsulates the state of the asynchronous invocation.

> In other Ice language mappings, the `AsyncResult` type is only used for asynchronous remote invocations. In JavaScript, an `AsyncResult` object can also be returned by methods of local Ice run time objects such as `Communicator` and `ObjectAdapter` when those methods have the potential to block or internally make remote invocations.

An instance of `AsyncResult` defines the following properties:

- `communicator`
  The communicator that sent the invocation.

- `connection`
  This method returns the connection that was used for the invocation. Note that, for typical proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change unexpectedly. The `connection` property only returns a non-nil value when the `AsyncResult` object is obtained by calling `flushBatchRequests` on a `Connection` object.

- `proxy`
  The proxy that was used for the invocation, or nil if the `AsyncResult` object was not obtained via a proxy invocation.

- `adapter`
  The object adapter that was used for the invocation, or nil if the `AsyncResult` object was not obtained via an object adapter invocation.

- `operation`
  The name of the operation being invoked.

`AsyncResult` objects also support the following methods:

- `cancel()`
  This method prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. `can`

cel is a local operation and has no effect on the server. A canceled invocation is considered to be completed, meaning `isCompleted` returns true, and the result of the invocation is an `Ice.InvocationCanceledException`.

- `isCompleted()`
  This method returns true if, at the time it is called, the result of an invocation is available. Otherwise, if the result is not yet available, the method returns false.

- `isSent()`
  When you make a proxy invocation, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued or an exception occurred before the request could be sent, `isSent` returns false.

- `sentSynchronously()`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

- `throwLocalException()`
  This method throws the local exception that caused the invocation to fail. If no exception has occurred yet, `throwLocalException` does nothing.

As described in the next section, an `AsyncResult` object provides additional functionality, including support for success and failure callbacks, that it inherits from a base type.

Back to Top ^

`Promise` Type in JavaScript

The `Ice.AsyncResult` type actually derives from `Ice.Promise`, which in turn derives from the standard JavaScript Promise. As you'll see below, one especially useful feature of a promise is its support for chaining, which allows you to make a series of asynchronous calls while still maintaining code readability.

Basic Promise Concepts

A promise represents a value that might be available in the future. The object always starts out in the pending state, and then transitions to a fulfilled or rejected state via calls to its `resolve` and `reject` methods, respectively. This state transition occurs only once; any subsequent calls to `resolve` or `reject` are ignored. The initial caller of `resolve` or `reject` can supply an argument that the promise object will pass along to its success or failure callbacks. Any number of callbacks can be registered on a single promise, and they can be registered before or after the promise has been resolved. The `resolve` and `reject` methods are helper methods added in `Ice.Promise` and are just references to the `resolve` and `reject` callback functions passed to the promise executor.

JavaScript

```
Ice.Promise = class extends Promise
{
    constructor(cb)
    {
        let res, rej;
        super((resolve, reject) => {
                res = resolve;
                rej = reject;

                if(cb !== undefined)
                {
                    cb(resolve, reject);
                }
            });

        this.resolve = res;
        this.reject = rej;
     }
...
}
```

A promise invokes any registered callbacks at the time the promise is resolved or rejected. Callbacks registered after a promise has been settled are invoked immediately.

Back to Top ^

Getting Started with Promises

The Promise method you'll use most often is `then`, inherited from JavaScript's standard Promise object. It accepts two arguments representing the success and failure callbacks:

JavaScript

```
function then(success, failure) { ... }
```

Both arguments are optional. Here is a simple example:

JavaScript

```
let p = new Ice.Promise();
p.then(
    r => {
        let [str, i] = r;
        console.log("received " + str + " and " + i);
    },
    ex => {
        console.log("failed: " + ex);
    });

try
{
    // do something...
    p.resolve(["string arg", 5]);
}
catch(ex)
{
    p.reject(ex);
}
```

The call to `resolve` resolves the promise and supplies an argument that the promise passes to the success callback. Similarly, the call to `reject` passes along the exception caught by the caller.

As you can see, the success and fail callbacks typically must know what arguments to expect when the promise is resolved.

Back to Top ^

Chaining Promises

The convenience and power offered by the promise concept becomes clear when you need to execute a sequence of asynchronous actions. Consider this example:

JavaScript

```
var p = new Ice.Promise();
p.then(    // A
    () => {
        // Step 1...
    }
).then(    // B
    () => {
        // Step 2...
    }
).then(    // C
    () => {
        // Step 3...
    }
);

// None of this happens until...
p.resolve();
```

Each call to `then` returns a new promise that is automatically chained to the preceding one. The promise returned by `then` in A resolves when Step 1 completes. The successful completion of A causes Step 2 to execute, and the successful completion of Step 2 resolves the promise returned by `then` in B, triggering Step 3 to execute. Finally, the whole chain of events won't begin until the initial promise p resolves.

Back to Top ^

Handling Errors with Promises

Let's extend our previous example to include an error handler:

JavaScript

```
var p = new Ice.Promise();
p.then(    // A
    () => {
        // Step 1...
    }
).then(    // B
    () => {
        // Step 2...
    }
).catch(
    ex => {
        // handle errors
    }
);
```

Here we call the `catch(failure)` method to establish an exception handler at the end of the promise chain, which is equivalent to calling `then(undefined, failure)`. You'll notice that none of the intermediate steps defines its own failure callback, with the intention that all errors will be handled in one location. After a call to `p.resolve` triggers execution of the chain, an exception raised by any of these steps propagates through each subsequent promise in the chain until a failure callback is found. If instead the application calls `p.reject`, none of the steps would execute and only the exception handler would be called.

Now let's examine the behavior when an intermediate step defines a failure callback:

JavaScript

```
let p = new Ice.Promise();
p.then(
    () => {
        console.log("step 1");
    },
    () => {
        console.log("fail 1");
    }
).then(
    () => {
        console.log("step 2");
    }
).catch(
    (ex) => {
        console.log("error");
    }
);
p.reject();

// Outputs:
fail 1
step 2
```

The results might surprise you at first. If a failure callback completes without raising an exception, the promise is considered to have resolved successfully and therefore triggers the success callback of the next promise in the chain. On the other hand, if a failure callback raises an exception, it will propagate to the next failure callback in the chain. This behavior allows a failure callback to recover from error situations (if possible) and continue with the normal logic flow, or raise an exception and skip to the next error handler.

> If no failure callback is defined in a chain after the point at which an exception occurs, that exception will be silently ignored by some promise implementations. For this reason, we always recommend terminating a chain with an exception handler. Furthermore, you should not let any uncaught exceptions be thrown from your final exception handler.

Next we'll illustrate this point by throwing an exception:

JavaScript

```
let p = new Ice.Promise();
p.then(
    () => {
        console.log("step 1");
    },
    ex => {
        console.log("fail 1");
        throw ex;
    }
).then(
    () => {
        console.log("step 2");
    }
).catch(
    ex => {
        console.log("error");
    }
);
p.fail();

// Outputs:
fail 1
error
```

Here you can see that the exception raised in the first failure callback bypasses Step 2 and triggers the final exception handler.

Back to Top ^

Passing Values between Promises

The promise implementation forwards any value returned by a success or failure callback to the next success callback in the chain, as shown below:

JavaScript

```
let p = new Ice.Promise();
p.then(
    () => {
        return "World!";
    }
).then(
    arg => {
        console.log("Hello " + arg);
    }
);
p.resolve();

// Outputs:
Hello World!
```

Back to Top ^

Chaining Asynchronous Events with Promises

Now that we've discussed promise fundamentals, let's explore more realistic use cases. The primary purpose of promises is chaining together a sequence of asynchronous events such that the next step in the chain won't execute until the previous step has completed. Common examples in Ice applications include a series of proxy invocations in which the calls must be made sequentially in a certain order, and proxy invocations in which the results of a preceding call must be obtained before the next call can be made.

Consider the following example:

JavaScript

```
let p = new Ice.Promise();
p.then(
    () => {  // Step 1
        let promise = // do something that creates a promise...
        return promise;
    }
).then(
    () => {  // Step 2
        let promise = // do something else that creates a promise...
        return promise;
    }
).then(
    () => {  // Step 3
        console.log("all done");
    }
);

p.resolve();
```

Steps 1 and 2 each initiate some asynchronous action that creates a promise, and the steps return the promise object as the result of the callback. Normally the result value would be passed as the argument to the next success callback in the chain, as we discussed earlier. However, when a callback returns a promise, execution of the next success or failure callback becomes dependent on the completion of the returned promise. In the example above, calling `p.succeed` causes Step 1 to execute, but Step 2 won't execute until the promise returned by Step 1 resolves successfully.

Back to Top ^

Promise API

Instances of `Ice.Promise` extend the JavaScript Promise and add the following methods to it:

- `finally(fn)`
  If the function throws an exception or returns a rejected promise that becomes the rejection value, otherwise the value that was used to resolve the promise is used to resolve the promise returned by `finally`. See examples below.

- `delay(ms)`
  Returns a new promise whose success or failure callback is invoked after the specified delay (in milliseconds). The results of the previous promise in the chain are forwarded to the next promise in the chain, as if the delay promise was not present.

- `resolve(value)`
  Resolves this promise. The value is passed to the success callback of the next promise in the chain.

- `reject(value)`
  Resolves this promise as failed. The value is passed to the failure callback of the next promise in the chain.

Additionally, the `Promise` type defines the following functions:

- `try(fn)`
  Returns a new promise whose execution depends on the completion of the given function. This function allows you to write code similar to the traditional try/catch/finally blocks of synchronous code, as shown below:

JavaScript

```
Ice.Promise.try(
    () => {
        // perform asynchronous actions...

        // return a promise to the last action in the chain...
    }
).catch(
    ex => {
        // handle errors
    }
).finally(
    () => {
        // clean up when the last action completes
    }
);
```

The `try` function makes error handling especially convenient because the chained exception handler will be called if an exception is thrown directly by the `try` callback, or if the callback returns a promise that later fails. Note that you could also arrange the `exception` and `finally` blocks this way:

JavaScript

```
Ice.Promise.try(
    () => {
        // perform asynchronous actions...

        // return a promise to the last action in the chain...
    }
).finally(
    () => {
        // clean up when the last action completes
    }
).catch(
    ex => {
        // handle errors
    }
);
```

If the `try` callback fails with an exception, the `finally` callback will be invoked and the failure will propagate to the `exception` callback. Of course, the order in which the callbacks are invoked in an error situation differs from the previous example.

- `delay(ms, value)`
  Similar to the prototype method described earlier, this function returns a promise whose success callback is invoked after the specified delay (in milliseconds):

JavaScript

```
Ice.Promise.delay(100, "John").then(
    name => { // Invoked after 100ms
        console.log("Hi " + name);
    });
```

Back to Top ^

Completion Callbacks in JavaScript

The `AsyncResult` promise returned by a proxy invocation resolves when the remote operation completes successfully or fails. The semantics of the success and failure functions depend on the proxy's invocation mode:

- Twoway proxies
  The success or failure function executes after the Ice run time in the client receives a reply from the server.

- Oneway and datagram proxies
  The success function executes after the Ice run time passes the message off to the socket. The failure function will only be called if an error occurs while Ice is attempting to marshal or send the message.
- Batch proxies
  The success function executes after the Ice run time queues the request. The failure function will only be called if an error occurs while Ice is attempting to marshal the message.

For all asynchronous Ice APIs, the failure function receives two arguments: the exception that caused the failure, and a reference to the `AsyncResult` promise for the failed invocation.

The success callback parameters depend on the operation signature. If the operation has non-`void` return type, the first parameter of the success callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration. Ice adds a reference to the `AsyncResult` object associated with the request as the final parameter to every success callback. Recall the example we showed earlier:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The success callback for `op` will receive and array with all the parameters:

JavaScript

```
function handleSuccess(r)
{
   let [returnVal, outp1, outp2] = r;
   ...
}
```

The failure callback is invoked if the invocation fails because of an Ice run time or user exception. This callback function must have the following signature:

JavaScript

```
function handleAnException(ex)
```

For example, the code below shows how to invoke the `getName` operation:

JavaScript

```
let proxy = ...;
let empNo = ...;
proxy.getName(empNo).then(
    name => {
        console.log("Name of employee #" + empNo + " is " + name);
    }
).catch(
    ex => {
        console.log("Failed to get name of employee #" + empNo);
        console.log(ex);
    }
);
```

Let's extend the example add a call to `getAddress`:

JavaScript

```
let proxy = ...;
let empNo = ...;
proxy.getName(empNo).then(
    name => {
        console.log("Name of employee #" + empNo + " is " + name);
        return proxy.getAddress(empNo);
    }
```

```
).then(
    addr => {
        console.log("Address of employee #" + empNo + " is " + addr);
    }
).catch(
    ex => {
        console.log("failed for employee #" + empNo);
        console.log(ex);
    }
);
```

Notice here that the success callback for `getName` returns the `AsyncResult` promise for `getAddress`, which means the second success callback won't be invoked until `getAddress` completes.

Back to Top ^

Asynchronous Oneway Invocations in JavaScript

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the proxy function on a oneway proxy for an operation that returns values or raises a user exception, the proxy function throws an instance of `TwowayOnlyException`.

The callback functions look exactly as for a twoway invocation. The failure function is only called if the invocation raised an exception during the proxy function ("on the way out"), and the success function is called as soon as the message is accepted by the local network connection. The success function takes no arguments.

Back to Top ^

Asynchronous Batch Requests in JavaScript

You can invoke operations via batch oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the proxy function on a oneway proxy for an operation that returns values or raises a user exception, the proxy function throws an instance of `TwowayOnlyException`.

The callback functions look exactly as for a twoway invocation. The failure function is only called if the batch invocation raised an exception before being queued. The success function takes no arguments. The returned promise for a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. It can also be marked completed if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The asynchronous proxy method `ice_flushBatchRequests` initiates an immediate flush of any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is accessible via the `connection` property of `AsyncResult`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

All versions of `ice_flushBatchRequests` return a promise whose success callback executes as soon as the batch request message is accepted by the local network connection. The success function takes no arguments. Ice only invokes the failure callback if an error occurs while attempting to send the message.

Back to Top ^

See Also

- Request Contexts
- Batched Invocations

Previous

Next