# Writing an Ice Application with JavaScript

This page shows how to create an Ice client application with JavaScript.

On this page:

## Compiling a Slice Definition for JavaScript

The first step in creating our JavaScript application is to compile our Slice definition to generate JavaScript proxies. You can compile the definition as follows:

```
$ slice2js Printer.ice
```

The `slice2js` compiler produces a single source file, `Printer.js`, from this definition. The exact contents of the source file do not concern us for now — it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

## Writing a Client in JavaScript

The client code, in `Client.js`, is shown below in full:

**JavaScript**

```
var Ice = require("ice").Ice;
var Demo = require("./Printer").Demo;

var ic;

Ice.Promise.try(
    function()
    {
        ic = Ice.initialize();
        var base = ic.stringToProxy("SimplePrinter:default -p 10000");
        return Demo.PrinterPrx.checkedCast(base).then(
            function(printer)
            {
                return printer.printString("Hello World!");
            });
    }
).finally(
    function()
    {
        if(ic)
        {
            return ic.destroy();
        }
    }
).exception(
    function(ex)
    {
        console.log(ex.toString());
        process.exit(1);
    });
```

The program begins with `require` statements that assign modules from the Ice run time and the generated code to convenient local variables. (These statements are necessary for use with NodeJS. Browser applications would omit these statements and load the modules a different way.)

The program begins with a call to `Ice.Promise.try` to launch a chain of promises (or futures) that handles the asynchronous nature of Ice invocations with a structure that resembles synchronous code.

1. The function passed to `try` is executed immediately. The body of this function begins by calling `Ice.initialize` to initialize the Ice run time. The call to `initialize` returns an `Ice.Communicator` reference, which is the main object in the Ice run time.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo.PrinterPrx.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy of type `Demo::PrinterPrx`; otherwise, if the proxy denotes an interface of some other type, the call returns `null`.
4. The `checkedCast` function involves a remote invocation to the server, which means this function has asynchronous semantics and therefore it returns a new promise object.
5. We call `then` on the promise returned by `checkedCast` and supply a "success" function, meaning the code that's executed when `checkedCast` succeeds. This inner function accepts one argument, `printer`, representing a proxy to the newly-downcasted object, or `null` if the remote object doesn't support the `Printer` interface.
6. Inside the success function, we now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal. Again, `printString` is a remote invocation, and it returns a promise that the success function passes along as its own return value.
7. The `then` function also returns a new promise which our outer function passes back to `try`. This outer promise is chained to the promise associated with the `printString` invocation; the outer promise completes successfully if and when the `printString` invocation completes successfully.
8. The function passed to `finally` is executed after the `try` block has completed, whether or not it completes successfully. If we created a communicator in the `try` block, we destroy it here. Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results. The `destroy` function has asynchronous semantics, so we return its promise to ensure no subsequent code is executed until `destroy` completes.
9. Lastly, the function passed to `exception` is the default exception handler for this entire promise chain.

# Running the Client in JavaScript

The server must be started before the client. Since Ice for JavaScript does not currently include a complete server-side implementation, we need to use a server from another language mapping. In this case, we will use the C++ server:

```
$ server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ node Client.js
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Ice::ConnectionRefusedException
    ice_cause: "Error: connect ECONNREFUSED"
    error: "ECONNREFUSED"
```

Note that, to successfully run the client, NodeJS must be able to locate the Ice for JavaScript modules. See the Ice for JavaScript installation instructions for more information.

## See Also

- Client-Side Slice-to-Ruby Mapping
- IceGrid