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`.

Using Ice with NodeJS

The language mapping is the same whether you're writing applications for NodeJS or a browser, but the code style is different enough that we describe the two platforms separately.

Writing a NodeJS Client

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

JavaScript

```javascript
const Ice = require("ice").Ice;
const Demo = require("./generated/Printer").Demo;

(async function()
{
    let communicator;
    try
    {
        communicator = Ice.initialize();
        const base = communicator.stringToProxy("SimplePrinter:default -p 10000");
        const printer = await Demo.PrinterPrx.checkedCast(base);
        if(printer)
        {
            await printer.printString("Hello World!");
        }
        else
        {
            console.log("Invalid proxy");
        }
    }
    catch(ex)
    {
        console.log(ex.toString());
        process.exitCode = 1;
    }
    finally
    {
        if(communicator)
        {
            await communicator.destroy();
        }
    }
}());
```

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 then defines an asynchronous function, which allows us to use the `await` keyword in our code when making proxy invocations. Here are the notable aspects of this code:

1. The body of the 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. 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. We apply the `await` keyword to the promise to wait for the call to complete.
5. If `checkedCast` returns a non-null value, 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 we await.
6. The `finally` block is executed after the `try` block has completed, whether or not it completes successfully. If we successfully 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 await it to ensure no subsequent code is executed until `destroy` completes.

Running the NodeJS Client

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.

Using Ice in a Browser

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

JavaScript

```
(function(){

const communicator = Ice.initialize();

async function printString()
{
    try
    {
        setState(State.Busy);

        const hostname = document.location.hostname || "127.0.0.1";
        const proxy = communicator.stringToProxy(`SimplePrinter:ws -h ${hostname} -p 10000`);

        const printer = await Demo.PrinterPrx.checkedCast(proxy);
        if(printer)
        {
            await printer.printString("Hello World!");
        }
        else
        {
            $("#output").val("Invalid proxy");
        }
    }
    catch(ex)
    {
        $("#output").val(ex.toString());
    }
    finally
    {
        setState(State.Idle);
    }
}
const State =
{
    Idle: 0,
    Busy: 1
};

function setState(newState)
{
    switch(newState)
    {
        case State.Idle:
        {
            // Hide the progress indicator.
            $("#progress").hide();
            $("body").removeClass("waiting");
            // Enable the button
            $("#print").removeClass("disabled").click(printString);
            break;
        }
        case State.Busy:
        {
            // Clear any previous error messages.
            $("#output").val("");
            // Disable buttons.
            $("#print").addClass("disabled").off("click");
            // Display the progress indicator and set the wait cursor.
            $("#progress").show();
            $("body").addClass("waiting");
            break;
        }
    }
}

setState(State.Idle);
}());
```

Here are the notable aspects of this code:

1. The program 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. Next the program defines the asynchronous function `printString`, which serves as the callback function for a UI button press. The `async` qualifier allows us to use the `await` keyword when making proxy invocations.
3. The code uses a simple state machine to manage the UI elements. Before making a remote invocation, the function enters the "busy" state to update the UI elements.
4. 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:ws -h` *`hostname`* `-p 10000"`, where *`hostname`* is the document location. 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.)
5. The proxy returned by `stringToProxy` is of type `Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces. 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`.
6. 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. We apply the `await` keyword to the promise to wait for the call to complete.
7. If `checkedCast` returns a non-null value, 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 we await.
8. The `finally` block is executed after the `try` block has completed, whether or not it completes successfully, in order to reset the program's state to "idle".

Here are some snippets from the corresponding HTML code:

HTML

```
<script type="text/javascript" src="Ice.js">
<script type="text/javascript" src="Printer.js">
<script type="text/javascript" src="Client.js">
...
<!-- UI elements -->
<section role="main" id="body">
    <div class="row">
        <div class="large-12 medium-12 columns">
            <form>
                <div class="row">
                    <div class="small-12 columns">
                        <a href="#" class="button small" id="print">Print String</a>
                    </div>
                </div>
                <div class="row">
                    <div class="small-12 columns">
                        <textarea id="output" readonly></textarea>
                    </div>
                </div>
                <div id="progress" class="row hide">
                    <div class="small-12 columns left">
                        <div class="inline left icon"></div>
                        <div class="text">Sending Request...</div>
                    </div>
                </div>
            </form>
        </div>
    </div>
</section>
```

The three `script` elements load the Ice run time, the generated code, and the application code, respectively.

A similar example can be found in `js/Ice/minimal` in the `ice-demos` repository.

See Also

- Client-Side Slice-to-JavaScript Mapping
- IceGrid