Getting Started with iRules LX, Part 3: Coding & Exception Handling

Eric Flores, 2016-08-06

So far in this series, we have covered the conceptual overview, components, workflows, licensing and provisioning of iRules LX. In this article, we will actually get our hands wet with some iRules LX code!

TCL Commands

As mentioned in the first article, we still need iRules TCL to make a RPC to Node.js. iRules LX introduces three new iRules TCL commands -

- `ILX::init` - Creates a RPC handle to the plugin's extension. We will use the variable from this in our other commands.
- `ILX::call` - Does the RPC to the Node.js process to send data and receive the result (2 way communication).
- `ILX::notify` - Does an RPC to the Node.js process to send data only (1 way communication, no response).

There is one caveat to the `ILX::call` and `ILX::notify` commands: you can only send 64KB of data. This includes overhead used to encapsulate the RPC so you will need to allow about 500B-1KB for the overhead. If you need to send more data you will need to create code that will send data over in chunks.

TCL Coding

Here is a TCL template for how we would use iRules LX.

```
when HTTP_REQUEST {
    set ilx_handle [ILX::init "my_plgin" "ext1"]
    if {[catch {ILX::call $ilx_handle "my_method" $user $passwd} result]} {
       log local0.error  "Client - [IP::client_addr], ILX failure: $result"
       # Send user graceful error message, then exit event
       return
    }

    # If one value is returned, it becomes TCL string
    if {$result eq 'yes'} {
        # Some action
    }

    # If multiple values are returned, it becomes TCL list
    set x [ lindex $result 0]
    set y [ lindex $result 1]
}
```

So as not to overwhelm you, we'll break this down to bite size chunks.

Taking line 2 from above, we will use `ILX::init` command to create our RPC handle to extension `ext1` of our LX plugin `my_plugin` and store it in the variable `ilx_handle`. This is why the names of the extensions and plugin matter.

```
set ilx_handle [ILX::init "my_plgin" "ext1"]
```

Next, we can take the handle and do the RPC -

```
if {[catch {ILX::call $ilx_handle "my_method" $user $passwd} result]} {
   log local0.error  "Client - [IP::client_addr], ILX failure: $result"
   # Send user graceful error message with whatever code, then exit event
   return
}
```

You can see on line 3, we make the `ILX::call` to the extension by using our RPC handle, specifying the method `my_method` (our "remote procedure") and sending it one or more args (in this case `user` and `passwd` which we got somewhere else). In this example we have wrapped `ILX::call` in a `catch` command because it can throw an exception if there is something wrong with the RPC (Note: catch should actually be used on any command that could throw an exception, b64decode for example). This allows us to gracefully handle errors and respond back to the client in a more controlled manner. If `ILX::call` is successful, then the return of `catch` will be a 0 and the result of the command will be stored in the variable `result`. If ILX::call fails, then the return of `catch` will be a 1 and the variable `result` will contain the error message. This will cause the code in the `if` block to execute which would be our error handling.

Assuming everything went well, we could now start working with data in the variable `result`. If we returned a single value from our RPC, then we could process this data as a string like so -

```
# If one value is returned, it becomes TCL string
if {$result eq 'yes'} {
    # Some action
}
```

But if we return multiple values from our RPC, these would be in TCL list format (we will talk more about how to return multiple values in the Node.js coding section). You could use `lindex` or any suitable TCL list command on the variable `result` -

```
# If multiple values are returned, it becomes TCL list
set x [ lindex $result 0]
set y [ lindex $result 1]
```

Node.js Coding

On the Node.js side, we would write code in our index.js file of the extension in the workspace. A code template will load when the extension is created to give you a starting point, so you dont have to write it from scratch.

To use Node.js in iRules LX, we provide an API for receiveing and sending data from the TCL RPC. Here is an example -

```
var f5 = require('f5-nodejs');

var ilx = new f5.ILXServer();

ilx.addMethod('my_method', function (req, res) {
   // req.params() function is used to get values from TCL. Returns JS Array
   var user = req.params()[0];
   var passwd = req.params()[1];

   <DO SOMETHING HERE>

   res.reply('<value>'); // Return one value as string or number
   res.reply(['value1','value2','value3']); // Return multiple values with an Array
});
```

```
ilx.listen();
```

Now we will run through this step-by-step.

On line 1, you see we import the `f5-nodejs` module which provides our API to the ILX server.

```
var f5 = require('f5-nodejs');
```

On line 3 we instantiate a new instance of the `ILXServer` class and store the object in the variable `ilx`.

```
var ilx = new f5.ILXServer();
```

On line 5, we have our `addMethod` method which stores methods (our remote procedures) in our ILX server.

```
ilx.addMethod('my_method', function (req, res) {
   // req.params() function is used to get values from TCL. Returns JS Array
   var user = req.params()[0];
   var passwd = req.params()[1];

   <DO SOMETHING HERE>

   res.reply('<value>'); // Return one value as string or number
   res.reply(['value1','value2','value3']); // Return multiple values with an Array
});
```

This is where we would write our custom Node.js code for our use case. This method takes 2 arguments -

- Method name - This would be the name that we call from TCL. If you remember from our TCL code above in line 3 we call the method `my_method`. This matches the name we put here.
- Callback function - This is the function that will get executed when we make the RPC to this method. This function gets 2 arguments, the `req` and `res` objects which follow standard Node.js conventions for the request and response objects.

In order to get our data that we sent from TCL, we must call the `req.param` method. This will return an array and the number of elements in that array will match the number of arguments we sent in `ILX::call`. In our TCL example on line 3, we sent the variables `user` and `passwd` which we got somewhere else. That means that the array from `req.params` will have 2 elements, which we assign to variables in lines 7-8. Now that we have our data, we can use any valid JavaScript to process it.

Once we have a result and want to return something to TCL, we would use the `res.reply` method. We have both ways of using `res.reply` shown on lines 12-13, but you would only use one of these depending on how many values you wish to return. On line 12, you would put a string or number as the argument for `res.reply` if you wanted to return a single value. If we wished to return multiple values, then we would use an array with strings or numbers. These are the only valid data types for `res.reply`.

We mentioned in the TCL result that we could get one value that would be a string or multiple values that would be a TCL list. The argument type you use in res.reply is how you would determine that.

Then on line 16 we start our ILX server so that it will be ready to listen to RPC.

```
ilx.listen();
```

That was quick a overview of the F5 API for Node.js in iRules LX. It is important to note that F5 will only support using Node.js in iRules LX within the provided API.

A Real Use Case

Now we can take what we just learned and actually do something useful. In our example, we will take POST data from a standard HTML form and convert it to JSON. In TCL we would intercept the data, send it to Node.js to transform it to JSON, then return it to TCL to replace the POST data with the JSON and change the Content-Type header -

```
when HTTP_REQUEST {
    # Collect POST data
    if { [HTTP::method] eq "POST" }{
        set cl [HTTP::header "Content-Length"]
        HTTP::collect $cl
    }
}
when HTTP_REQUEST_DATA {
    # Send data to Node.js
    set handle [ILX::init "json_plugin" "json_ext"]
    if {[catch {ILX::call $handle "post_transform" [HTTP::payload]} json]} {
      log local0.error  "Client - [IP::client_addr], ILX failure: $json"
      HTTP::respond 400 content "<html>Some error page to client</html>"
      return
    }

    # Replace Content-Type header and POST payload
    HTTP::header replace "Content-Type" "application/json"
    HTTP::payload replace 0 $cl $json
}
```

In Node.js, would only need to load the built in module `querystring` to parse the post data and then `JSON.stringify` to turn it into JSON.

```
'use strict'
var f5 = require('f5-nodejs');
var qs = require('querystring');

var ilx = new f5.ILXServer();

ilx.addMethod('post_transform', function (req, res) {
  // Get POST data from TCL and parse query into a JS object
  var postData = qs.parse(req.params()[0]);

  // Turn postData into JSON and return to TCL
  res.reply(JSON.stringify(postData));
});

ilx.listen();
```

Note: Keep in mind that this is only a basic example. This would not handle a POST that used 100 continue or mutlipart POSTs.

Exception Handling

iRules TCL is very forgiving when there is an unhanded exception. When you run into an unhandled runtime exception (such as an invalid base64 string you tried to decode), you only reset that connection. However, Node.js (like most other programming languages) will crash if you have an unhandled runtime exception, so you will need to put some guard rails in your code to avoid this. Lets say for example you are doing `JSON.parse` of some JSON you get from the client. Without proper exception handling any client could crash your Node.js process by sending invalid JSON. In iRules LX if a Node.js process crashes 5 times in 60 seconds, BIG-IP will not attempt to restart it which opens up a DoS attack vector on your application (max restarts is user configurable, but good code eliminates the need to change it). You would have to manually restart it via the Web UI or TMSH.

In order to catch errors in JavaScript, you would use the try/catch statement. There is one caveat to this: code inside a try/catch statement is not optimized by the v8 complier and will cause a significant decrease in performance. Therefore, we should keep our code in try/catch to a minimum by putting only the functions that throw exceptions in the statement. Usually, any function that will take user provided input can throw.

Note: The subject of code optimization with v8 is quite extensive so we will only talk about this one point. There are many blog articles about v8 optimization written by people much smarter than me. Use your favorite search engine with the keywords v8 optimization to find them.

Here is an example of try/catch with `JSON.parse` -

```
ilx.addMethod('my_function', function (req, res) {
  try {
    var myJson = JSON.parse(req.params()[0])  // This function can throw
  } catch (e) {
    // Log message and stop processing function
    console.error('Error with JSON parse:', e.message);
    console.error('Stack trace:', e.stack);
    return;
  }

  // All my other code is outside try/catch
  var result = ('myProperty' in myJson) ? true : false;
  res.reply(result);
});
```

We can also work around the optimization caveat by hoisting a custom function outside try/catch and calling it inside the statement -

```
ilx.addMethod('my_function', function (req, res) {
  try {
    var answer = someFunction(req.params()[0]) // Call function from here that is defined on line 16
  } catch (e) {
    // Log message an stop processing function
    console.error('Error with someFunction:', e.message);
    console.error('Stack trace:', e.stack);
    return;
  }

  // All my other code is outside try/catch
  var result = (answer === 'hello') ? true : false;
  res.reply(result);
});

function someFuntion (arg) {
  // Some code in here that can throw
```

```
  return result
}
```

RPC Status Return Value

In our examples above, we simply stopped the function call if we had an error but never let TCL know that we encountered a problem. TCL would not know there was a problem until the `ILX::call` command reached it's timeout value (3 seconds by default). The client connection would be held open until it reached the timeout and then reset. While it is not required, it would be a good idea for TCL to get a return value on the status of the RPC immediately. The specifics of this is pretty open to any method you can think of but we will give an example here.

One way we can accomplish this is by the return of multiple values from Node.js. Our first value could be some type of RPC status value (say an RPC error value) and the rest of the value(s) could be our result from the RPC. It is quite common in programming that make an error value would be 0 if everything was okay but would be a positive integer to indicate a specific error code.

Here in this example, we will demonstate that concept. The code will verify that the property `myProperty` is present in JSON data and put it's value into a header or send a 400 response back to the client if not.

THe Node.js code -

```
ilx.addMethod('check_value', function (req, res) {
  try {
    var myJson = JSON.parse(req.params()[0])  // This function can throw
  } catch (e) {
    res.reply(1); //<---------------------------------- The RPC error value is 1 indicating invalid J
    return;
  }

  if ('myProperty' in myJson){
    // The myProperty property was present in the JSON data, evaluate its value
    var result = (myJson.myProperty === 'hello') ? true : false;
    res.reply([0, result]);   //<-------------------------- The RPC error value is 0 indicating succe
  } else {
    res.reply(2);   //<-------------- The RPC error value is 2 indicating myProperty was not present
  }
});
```

In the code above the first value we return to TCL is our RPC error code. We have defined 3 possible values for this -

- 0 - RPC success
- 1 - Invalid JSON
- 2 - Property "myProperty" not present in JSON

One our TCL side we would need to add logic to handle this value -

```
when HTTP_REQUEST {
  # Collect POST data
  if { [HTTP::method] eq "POST" }{
    set cl [HTTP::header "Content-Length"]
    HTTP::collect $cl
  }
}

when HTTP_REQUEST_DATA {
```

```
  # Send data to Node.js
  set handle [ILX::init "json_plugin" "json_checker"]
  if {[catch [ILX::call $handle "check_value" [HTTP::payload]] json]} {
    log local0.error  "Client - [IP::client_addr], ILX failure: $result"
    HTTP::respond 400 content "<html>Some error page to client</html>"
    return
  }

  # Check the RPC error value
  if {[lindex $json 0] > 0} {
    # RPC error value was not 0, there is a problem
    switch [lindex $json 0] {
      1 { set error_msg "Invalid JSON"}
      2 { set error_msg "myProperty property not present"}
    }
    HTTP::respond 400 content "<html>The following error occured: $error_msg</html>"
  } else {
    # If JSON was okay, insert header with myProperty value
    HTTP::header insert "X-myproperty" [lindex $json 1]
  }
}
```

As you can see on line 19, we check the value of the first element in our TCL list. If it is greater than 0 then we know we have a problem. We move on down further to line 20 to determine the problem and set a variable that will become part of our error message back to the client.

Note: Keep in mind that this is only a basic example. This would not handle a POST that used 100 continue or mutlipart POSTs.

In the next article in this series, we will cover how to install a module with NPM and some best practices.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc. Corporate Headquarters info@f5.com	F5 Networks Asia-Pacific apacinfo@f5.com	F5 Networks Ltd. Europe/Middle-East/Africa emeainfo@f5.com	F5 Networks Japan K.K. f5j-info@f5.com