Getting Started with iRules LX, Part 5: Troubleshooting

Eric Flores, 2016-10-06

When you start writing code, you will eventually run into some issue and need to troubleshoot. We have a few tools available for you.

Logging

One of the most common tools use for troubleshooting is logging. STDOUT and STDERR for the Node.js processes end up in the file `/var/log/ltm`. By temporarily inserting functions such as `console.log`, `console.error`, etc. you can easily check the state of different objects in your code at run time.

There is one caveat to logging in iRules LX; logging by most of most processes on BIG-IP is throttled by default. Basically, if 5 messages from a process come within 1 second, all other messages with be truncated. For Node.js, every line is a separate message so something like a stack trace will get automatically truncated. To remove this limit we can change the DB variable `log.sdmd.level` to the value `debug` in TMSH as such -

```
root@(test-ve)(cfg-sync Standalone)(Active)(/Common)(tmos)
# modify sys db log.sdmd.level value debug
```

SDMD is the process that manages the Node.js instances. This turns on verbose logging for SDMD and all the Node.js processes so you will want to turn this back to `info` when you are finished troubleshooting.

Once that is done you can start adding logging statements to your code. Here, I added a line to tell me when the process starts (or restarts) -

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

console.log('Node.js process starting.');

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

The logs would then show me this -

```
Jun  7 13:26:08 test-ve info sdmd[12187]: 018e0017:6: pid[21194] plugin[/Common/test_pl1.ext1] Node.j
```

Node Inspector

BIG-IP comes packaged with Node Inspector for advanced debugging. We won't cover how to use Node Inspector because most developers skilled in Node.js already know how to use it and there are already many online tutorials for those that dont.

Note: We highly recommend that you do not use Node Inspector in a production environment. Debuggers lock the execution of code at break points pausing the entire application, which does not work well for a realtime network device. Ideally debugging should only be done on a dedicated BIG-IP (such as a lab VE instance) in an isolated development environment.

Turn on Debugging

The first thing we want to do is to put our extension into single concurrency mode so that we are only working with one Node.js process while debugging. Also, while we are doing that we can put the extension into debugging mode. This can be done from TMSH like so -

```
root@(test-ve)(cfg-sync Standalone)(Active)(/Common)(tmos)
# list ilx plugin DC_Plugin
ilx plugin DC_Plugin {
    disk-space 156
    extensions {
        dc_extension { }
        dc_extension2 { }
    }
    from-workspace DevCentralRocks
    staged-directory /var/ilx/workspaces/Common/DevCentralRocks
}
root@(test-ve)(cfg-sync Standalone)(Active)(/Common)(tmos)
# modify ilx plugin DC_Plugin extensions { dc_extension { concurrency-mode single command-options add
root@(test-ve)(cfg-sync Standalone)(Active)(/Common)(tmos)
# list ilx plugin DC_Plugin
ilx plugin DC_Plugin {
    disk-space 156
    extensions {
        dc_extension {
            command-options { --debug }
            concurrency-mode single
        }
        dc_extension2 { }
    }
    from-workspace DevCentralRocks
    staged-directory /var/ilx/workspaces/Common/DevCentralRocks
}
root@(test-ve)(cfg-sync Standalone)(Active)(/Common)(tmos)
# restart ilx plugin DC_Plugin immediate
```

Increase ILX::call Timeout

Also, if you recall from part 3 of this series, the TCL command `ILX::call` has a 3 second timeout by default that if there is no response to the RPC, the command throws an exception. We don't need to worry about this for the command `ILX::notify` because it does not expect a response from the RPC. While we have the debugger paused on a break point you are pretty much guaranteed to hit this timeout, so we need to increase it to allow for our whole method to finish execution. To do this, you will need to add arguments to the `ILX::call` highlighted in red -

```
ILX::call $ilx_handle -timeout big_number_in_msec method arg1
```

This number should be big enough to walk through the entire method call while debugging. If it will take you 2 minutes, then the value should be 120000. Once you update this make sure to reload the workspace.

Launching Node Inspector

Once you have put Node.js into debugging mode and increased the timeout, you now you can fire up Node Inspector. We will need to find the debugging port that our Node.js process has been assigned via TMSH -

```
root@(eflores-ve3)(cfg-sync Standalone)(Active)(/Common)(tmos)
# show ilx plugin DC_Plugin
```

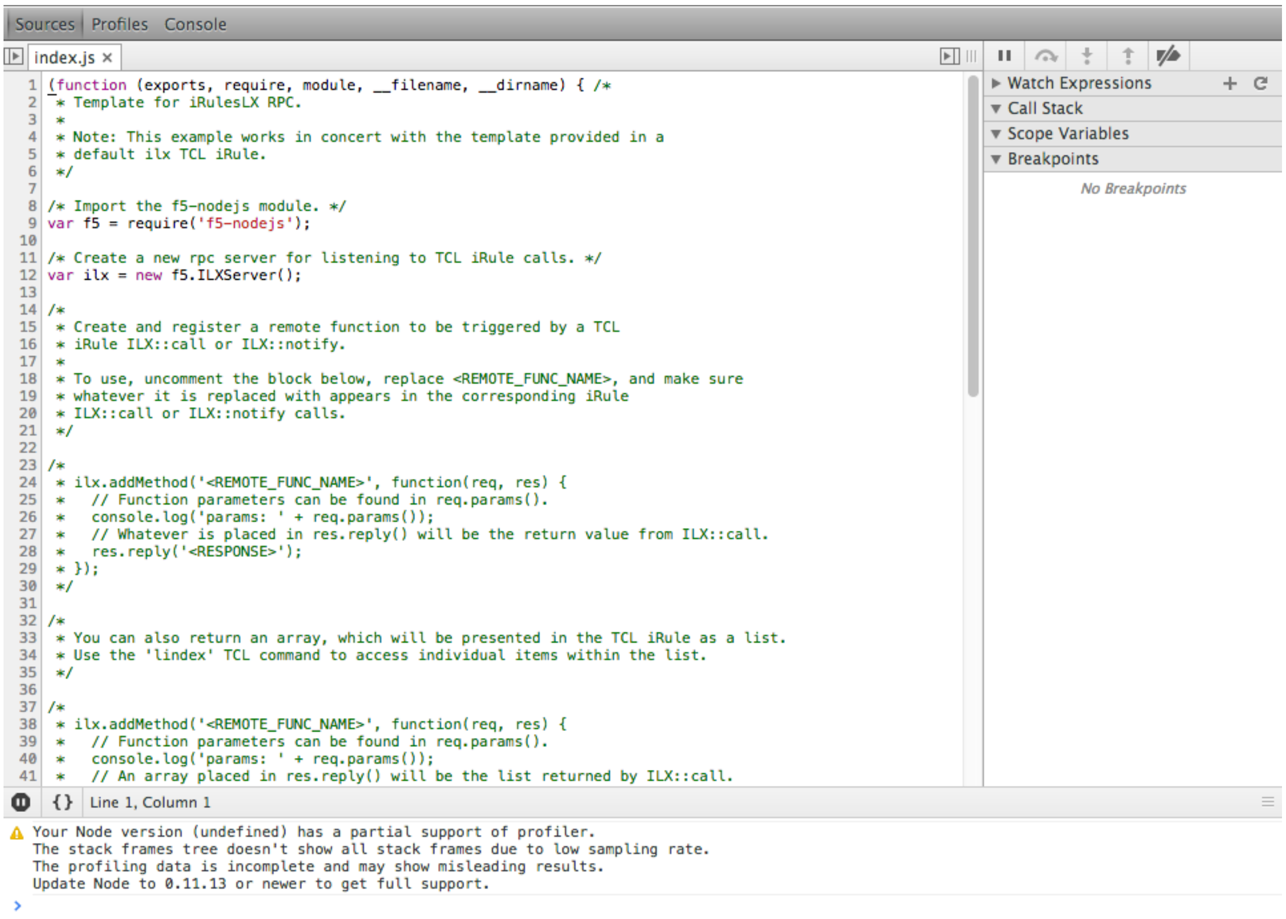
```
<----------------snipped ----------------->


     ---------------------------------
     | Extension Process: dc_extension
     ---------------------------------
     | Status                 running
     | PID                    25975
     | TMM                    0
     | CPU Utilization (%)    0
     | Debug Port             1033
     | Memory (bytes)
     |   Total Virtual Size   1.2G
     |   Resident Set Size    5.8K
```

Now that we have the debugging port we can put that and the IP of our management interface into the Node Inspector arguments -

```
[root@test-ve:Active:Standalone] config # /usr/lib/node_modules/.bin/node-inspector --debug-port 1033
Node Inspector v0.8.1
Visit http://192.168.0.245:8080/debug?port=1033 to start debugging.
```

We simply need to take the URL given to us and put it in our web browser. Then we can start debugging away!



```
(function (exports, require, module, __filename, __dirname) { /*
 * Template for iRulesLX RPC.
 *
 * Note: This example works in concert with the template provided in a
 * default ilx TCL iRule.
 */

/* Import the f5-nodejs module. */
var f5 = require('f5-nodejs');

/* Create a new rpc server for listening to TCL iRule calls. */
var ilx = new f5.ILXServer();

/*
 * Create and register a remote function to be triggered by a TCL
 * iRule ILX::call or ILX::notify.
 *
 * To use, uncomment the block below, replace <REMOTE_FUNC_NAME>, and make sure
 * whatever it is replaced with appears in the corresponding iRule
 * ILX::call or ILX::notify calls.
 */

/*
 * ilx.addMethod('<REMOTE_FUNC_NAME>', function(req, res) {
 *   // Function parameters can be found in req.params().
 *   console.log('params: ' + req.params());
 *   // Whatever is placed in res.reply() will be the return value from ILX::call.
 *   res.reply('<RESPONSE>');
 * });
 */

/*
 * You can also return an array, which will be presented in the TCL iRule as a list.
 * Use the 'lindex' TCL command to access individual items within the list.
 */

/*
 * ilx.addMethod('<REMOTE_FUNC_NAME>', function(req, res) {
 *   // Function parameters can be found in req.params().
 *   console.log('params: ' + req.params());
 *   // An array placed in res.reply() will be the list returned by ILX::call.
```

```
Your Node version (undefined) has a partial support of profiler.
The stack frames tree doesn't show all stack frames due to low sampling rate.
The profiling data is incomplete and may show misleading results.
Update Node to 0.11.13 or newer to get full support.
```

Conclusion

We hope you have found the **Getting Started with iRules LX** series very informative and look forward to hearing about how iRules LX has helped you solved problems. If you need any help with using iRules LX, please dont hesitate to ask a question on DevCentral.