F5 iRules Data Plane Programmability Documentation

F5 Networks, Inc.

Feb 14, 2020

Agility 2020 Hands-on Lab Guide

Contents:

1 Getting Started

Please follow the instructions provided by the instructor to start your lab and access your jump host.

Note: All work for this lab can be performed via an RDP connection to the Windows jumphost, or by selecting bigip01 and accessing the system via the TMUI web interface.

1.1 Lab Topology

The following components have been included in your lab environment:

- 1 x F5 BIG-IP VE
- 1 x Linux LAMP Webserver
- 1 x Windows Jumphost

2

Creating and Implementing an LX iRule

In this lab we will learn how to use iRules LX with a basic example to introduce you to the concepts of iRules LX and their configuration objects. We will have a web application that has a web form. When we submit the form, the page will display our POST data. As part of the lab exercise, we will apply an LX iRule that will convert the form POST data into JSON and change the Content-Type header.

The practicality of this use case could be when you have some type of legacy front end service that can only POST data as a standard query string, but a new back end service takes data as JSON. It would be pretty impractical to use tradition iRules to perform the translation. However, this task is trivial for iRules LX because of the power of Node.js. We will implement an LX iRule that will accomplish this.

2.1 Lab 1 - Creating and Implementing an LX iRule

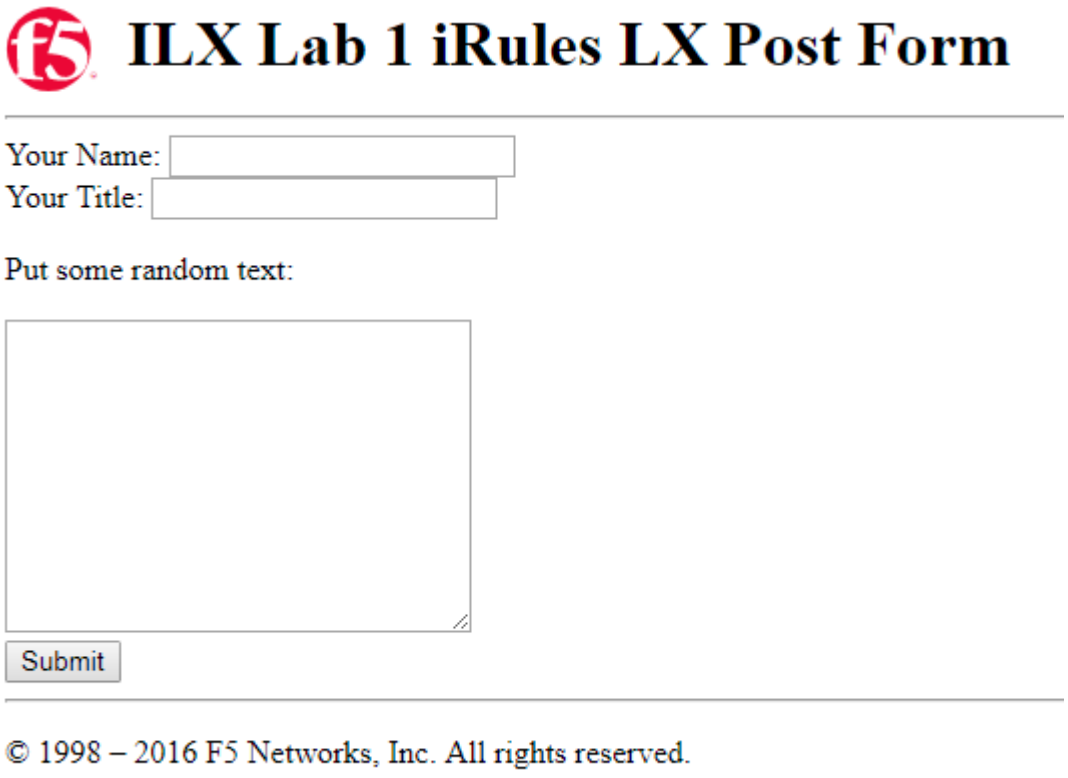
2.1.1 Test and Review the Existing Configuration

Important:

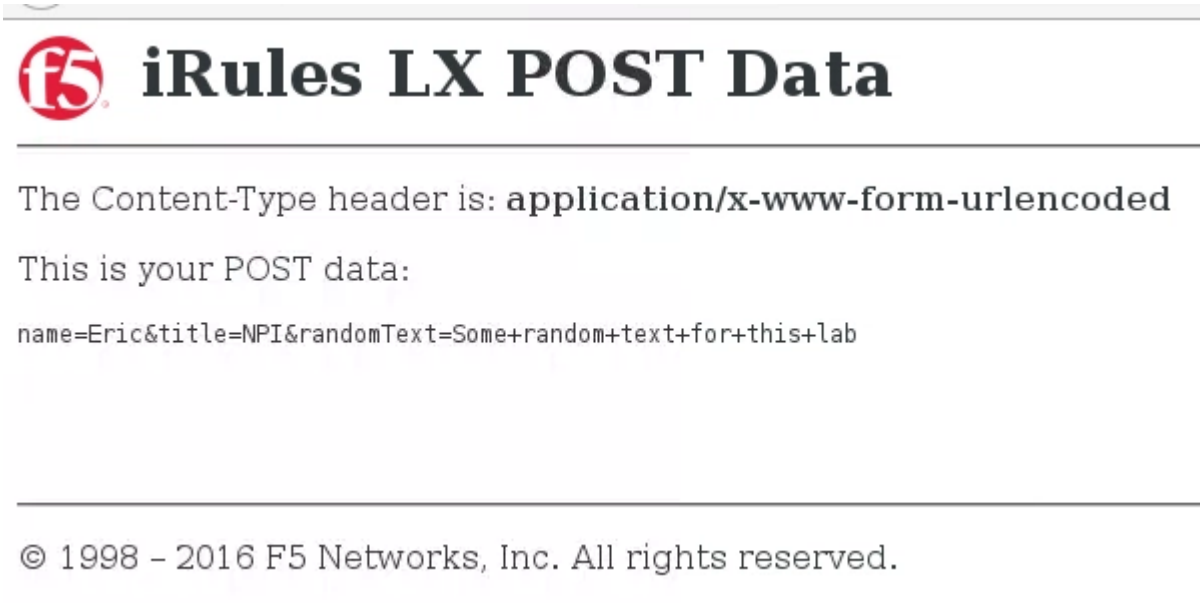
- All code snippets are stored on the Windows Server 2016 Jumphost within a folder titled **ilxcode**.

To start off we have a web application that has a web form that we enter some information into and submit.

1. Lets look at the web app at the URL **http://10.1.20.20/ilxlab1/** (Lab 1 on bookmarks).
2. The response of the **POST** will show our form data and **"Content-Type"** header.
3. Here is the example of the web form –



4. Go ahead and run your own test of the web app. Observe the "**Content-Type**" header and **POST** data values.
5. Here is an example of the response to a POST -



2.1.2 Create the LX Workspace

The first thing we need to do is create an LX Workspace.

1. On the BIG-IP, navigate over to the LX workspaces menu in the tab located at **Local Traffic > iRules > LX Workspaces.**
2. Then select the **create** button at the top right of the table and name the workspace **ilxlab1**. You will now have an empty workspace.

2.1.3 Create the Extension

Next, we create a new extension (the Node.js code that will run). The name of the extension will matter later because we will call that from our iRules TCL code.

1. To create the extension, click the **Add Extension** button at the bottom of the editor, then give it the name **ilxlab1_ext**.
2. The various files of the extension will show up. Select the **index.js** file and you should see a template of example code in the editor window. Normally you could use this example code as a starting point, but in our case we should delete all the example code from the window.
3. In the Atom editor, locate the file named **ilxlab1.js** within the ilxcode folder and double click it which should open it in a text editor.
4. Copy and paste this into the **index.js** file on our BIG-IP.
5. Then you will need to save the changes to this file with the **Save File** button at the bottom of the editor window.

2.1.4 Create the TCL iRule

Next, we need to create the TCL iRule that will call our Node.js code.

1. Click the button **Add iRule** at the bottom of the editor window
2. Name the iRule **json_post** and don't check the box to include example code (we don't need the example code for this lab).
3. In the Atom editor, locate the file named within the ilxcode folder called **ilxlab1.tcl**.
4. Copy and paste the contents into the **json_post** iRule file.
5. Then you will need to save the changes to this file with the **Save File** button at the bottom of the editor window.

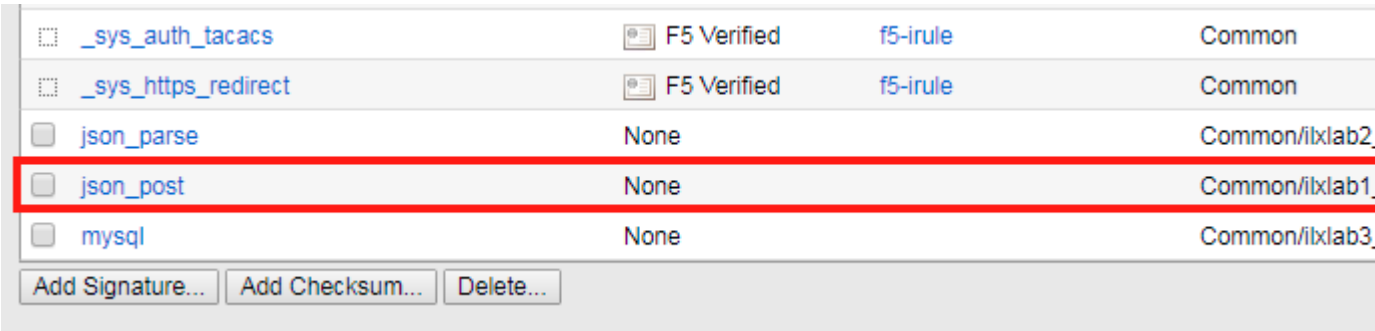
2.1.5 Create the LX Plugin

1. Now that we have our code in a workspace, you will need to navigate over to the LX Plugins menu in the tab located at **Local Traffic > iRules > LX Plugins**
2. Click the **Create** button, name the plugin **ilxlab1_pl**
3. Select the **ilxlab1** workspace and click **Finished**. This makes the Node.js code active.

2.1.6 Apply the LX iRule to the Virtual Server

Now that we have our Node.js code running, we can put it to use. In order to use the code from the plugin we must assign the TCL iRule to a virtual server.

1. Just so we can be familiar with it (but it is not required), we will look for the TCL iRule in the **Local Traffic > iRules > iRules List** menu.
2. You will find the iRule that we created in the workspace located there with a Partition/Path that has the same name as our plugin.

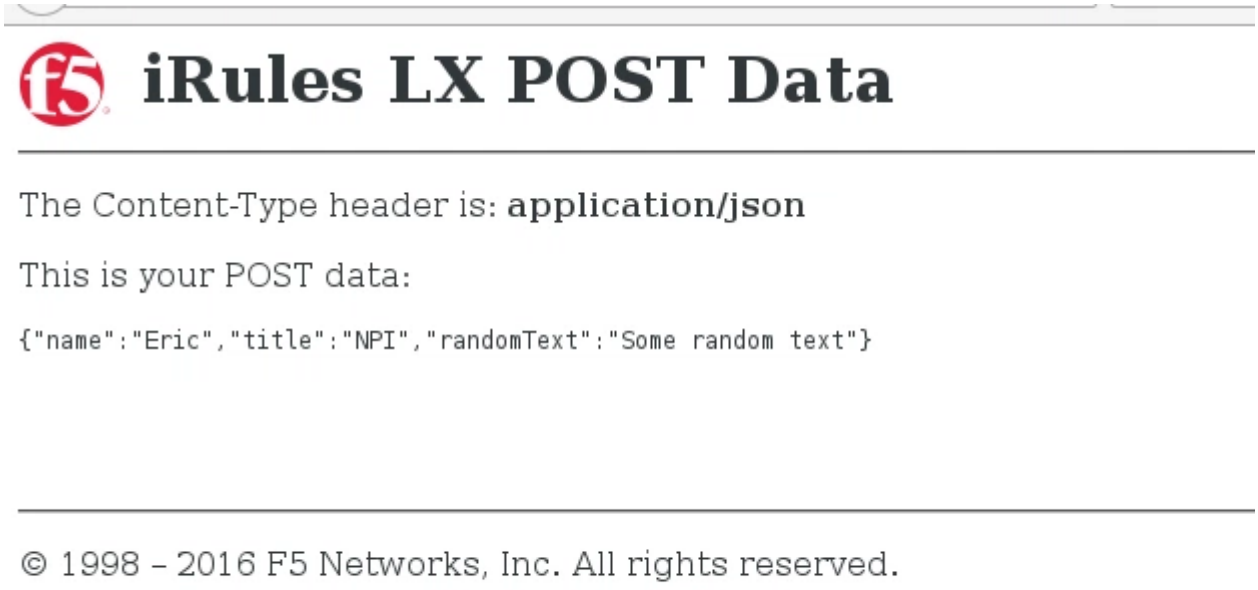


3. You wont be able to make changes from here. This is the same behavior as an iApp with strict updates enabled.
4. Now navigate over to our virtual server list, click the **Edit** button (under the **resources** column) for the virtual **ilxlab1_vs** and select the **Manage** button for iRules.
5. If you scroll to the bottom of the available iRules list, you should see the iRule from our plugin.

6. Move this iRule to the over to the enabled section and click **finished**.

2.1.7 Testing the LX iRule

1. Now let's navigate to the second tab on the browser with the web page of our app.
2. Go back to the web form and submit the information again. You will see now that the data has been converted to JSON and the **Content-Type** header has been changed.



As you can see, with iRules LX we can implement solutions with very few lines of code. If we wanted to accomplish the same goal in TCL alone, it would most likely take several hundred lines of code.

2.1.8 Workspace Package Management

Lastly, we will show package management for LX workspaces. While it is fairly simple to move TCL iRules from a dev/test environment to production because it is a single file, iRules LX can have an almost unlimited number of files depending on how many NPM modules a solution needs. Therefore, workspaces have been given the ability to export and import packages as a tgz file to have a more convenient method of transporting iRules LX code. In this exercise, we will export our package and import it back into the same device (but normally import would happen on a separate BIG-IP).

Export/Import a Workspace

1. Go to the **LX Workspaces** list, check the box of our *ilxlab1* workspace and click the **Export** button below the list. This will save the file to the user's **Downloads** folder.
2. Now click the **Import** button on the top right hand corner of the workspace list.
3. On the next window give the imported workspace the name of **ilxlab1_restore**
4. select the option **Archive File**, and use the **Choose File** button to find the tgz file in the user's **Downloads** folder.
5. When you click the **Import** button you will be taken back to the workspace list and you should see the imported workspace now. Feel free to navigate into the imported workspace.

You have concluded lab exercise #1

3 Asynchronous Programming

In this lab we will demonstrate the concept of asynchronous programing with a LX iRule that will do queries to a MySQL database. For this exercise, we will be using the file `ilxlab3_steps.js` to cut and paste code into the BIG-IP.

3.1 Lab 1 - Asynchronous Programming

3.1.1 Test and Review the Existing Configuration

In this lab we will be working with the virtual server **(10.1.20.22)** & workspace named **ilxlab3**. The plugin and TCL iRule are already assigned to the virtual server. To start off we have a web application that displays a list of users in a database. This web app is configured on our BIG-IP at the URL http://10.1.20.22/.

3.1.2 SQL Database Lookup

In this lab we are simply going to view some log statements into the Node.js and look at the order they appear in the log file.

1. First we will review the sql query method in our extension code highlighted below:

```
// Add a method
ilx.addMethod('get_users', function(req, res) {
  // Perform the query from pool
  sqlPool.query(
    'SELECT id, name, grp FROM users_db.users ORDER BY id;',
    function(err, rows) {
      if (err) {
        // MySQL query failed for some reason, send a 2 back to TCK
        console.error('Error with query: ', err.message);
        return res.reply(2);
      }

      // Check array length from sql
      if (rows.length)
        res.reply([0, rows]);
      else
```

(continues on next page)

(continued from previous page)

```
        res.reply(1); // if 0 return 1 to the Tcl iRule to show no matching records
    }
  );
});
```

You will notice that the function has 2 arguments, the first being the text of the actual query. Because this method is asynchronous, the second argument is the callback function that will get executed when the query answer is received by Node.js.

2. To demonstrate asynchronous behavior, we will put logging statements before and after the query method as such:

Code Step 1

```
// Add a method
ilx.addMethod('get_users', function(req, res) {
  // Perform the query from pool
  console.log('Starting SQL query');
  sqlPool.query(
    'SELECT id, name, grp FROM users_db.users ORDER BY id;',
    function(err, rows) {
      if (err) {
        // MySQL query failed for some reason, send a 2 back to TCK
        console.error('Error with query: ', err.message);
        return res.reply(2);
      }
      console.log('There are', rows.length,'records in the DB.');

      // Check array length from sql
      if (rows.length)
        res.reply([0, rows]);
      else
        res.reply(1); // if 0 return 1 to the Tcl iRule to show no matching records
    }
  );
  console.log('SQL query finished.');
});
```

Make sure to use the TMSH plugin restart command after you reload the workspace.

1. Now tail the log contents of the log file with the following BASH command and then refresh the ilxlab3 web page:

   ```
   # tail -f /var/log/ilx/Common.ilxlab3_pl.mysql
   ```

2. What do you notice about the order of the log statements?

3. Now let's make the following changes to the node.js as seen below.

Code Step 2

```
// Add a method
ilx.addMethod('get_users', function(req, res) {
  // Perform the query from pool
  console.log('Starting SQL query');
  sqlPool.query(
    'SELECT id, name, grp FROM users_db.users ORDER BY id;',
```

(continues on next page)

(continued from previous page)

```
    function(err, rows) {
      if (err) {
        // MySQL query failed for some reason, send a 2 back to TCK
        console.error('Error with query: ', err.message);
        return res.reply(2);
      }
      console.log('There are', rows.length,'records in the DB.');

      // Check array length from sql
      if (rows.length)
        res.reply([0, rows]);
      else
        res.reply(1); // if 0 return 1 to the Tcl iRule to show no matching records
      console.log('SQL query is really finished.');
    }
  );
  console.log('Function call is finished.');
});
```

1. Use the TMSH plugin restart command after you reload the workspace. Now tail the log contents of the log file again and then refresh the **ilxlab3** web page. You will see that they are in the right order. The callback function is executed much later because I/O responses take much longer.

   But you might ask, how much later is the callback function executing?

2. To answer that question, lets add some more code:

   Code Step 3

```
// Add a method
ilx.addMethod('get_users', function(req, res) {
  // Perform the query from pool
  console.log('Starting SQL query');
  var start = Date.now();
  sqlPool.query(
    'SELECT id, name, grp FROM users_db.users ORDER BY id;',
    function(err, rows) {
      if (err) {
        // MySQL query failed for some reason, send a 2 back to TCK
        console.error('Error with query: ', err.message);
        return res.reply(2);
      }
      console.log('There are', rows.length,'records in the DB.');

      // Check array length from sql
      if (rows.length)
        res.reply([0, rows]);
      else
        res.reply(1); // if 0 return 1 to the Tcl iRule to show no matching records
      console.log('SQL query is really finished, time:', Date.now() - start, 'msec');
    }
  );

  console.log('Function call is finished.');
});
```

1. Use the TMSH plugin restart command after you reload the workspace. Now tail the log contents of the log file again and then refresh the ilxlab3 web page. Most likely, you are seeing that the time logged is in the order of tens of milliseconds. As you saw from the I/O time table in the presentation, this is an eternity compared to reads from local memory.

4 iRules LX Streaming

In this lab exercise, you will learn how to create LX plugins that can be use in streaming or HTTP mode. In the interest of time, we will taking existing workspaces then and take the code to a full working configuration on a virtual server. We will be using the virtual server ilxlab4_stream_vs (10.0.0.23).

4.1 Lab 1 - iRules LX Streaming

4.1.1 Creating and Implementing a Streaming LX Plugin

In this lab we will be loading an LX plugin in streaming mode. To keep the lab simple, we will only be loading a plugin that will print the client data to hexdump format in the log files.

Review the LX Workspace and Install NPM package

The first thing we need to do is view the LX Workspace. On the desktop, navigate over to the LX workspaces menu in the tab located at **Local Traffic > iRules > LX Workspaces**. Then click the workspace named **ilxlab4_stream**. You should see an extension named hexdump, then click on the **index.js** file. Also, we should note that you will not see a TCL rule in the workspace.

Just for reference, here is the Node.js code below:

```
'use strict';
// Hexdump all client data to stdout on L4 virtual server
var f5 = require('f5-nodejs');
var plugin = new f5.ILXPlugin();
var hexy = require('hexy');

// Register a listener for the client ILXPlugin "connect" event
plugin.on('connect', function(flow) {
  // Register a listener for the ILXStream "data" event
  flow.client.on('data', function (data) {
    console.log(hexy.hexy(data)); //Print the client data to STDOUT
    flow.server.write(data);  //Pass the client data to the server stream
  })

  // Create event listeners for error events
```

(continues on next page)

(continued from previous page)

```
  flow.client.on('error', function(errorText) {
    console.log('client error event: ' + errorText);
  });
  flow.server.on('error', function(errorText) {
    console.log('server error event: ' + errorText);
  });
  flow.on('error', function(errorText) {
    console.log('flow error event: ' + errorText);
  });
});

// Tell TMM not to send data from server to Node
var options = new f5.ILXPluginOptions();
options.handleServerData = false;
plugin.start(options); //Start the plugin in streaming mode
```

As you can see from the code above we are loading the hexy package for doing the hexdumps of the buffer chunk. Therefore, we need to install this package into the workspace.

1. To do this you will need to SSH to the BIG-IP and execute the following commands from the BASH prompt:

```
# cd /var/ilx/workspaces/Common/ilxlab4_stream/extensions/hexdump/
# npm install --save hexy
```

Create the LX Plugin

1. With our code already in a workspace, you will need to navigate over to the LX Plugins menu in the tab located at **Local Traffic > iRules > LX Plugins**. Click the *Create* button, name the plugin *ilxlab4_stream_pl*, select the **ilxlab4_stream** workspace and click finish to save the changes.
2. We still need to configure a few more things so once you are back to the LX Plugin list, click on the **ilxlab4_stream_pl** plugin and then click on the **hexdump** extension. Change the following settings:

Setting	New Value	Reason
Concurrency Mode	Single	Keep logs for all connections in a single file.
iRules LX Logging	Checked	Will make extension send logs to dedicated file.

Create the iRules LX Profile

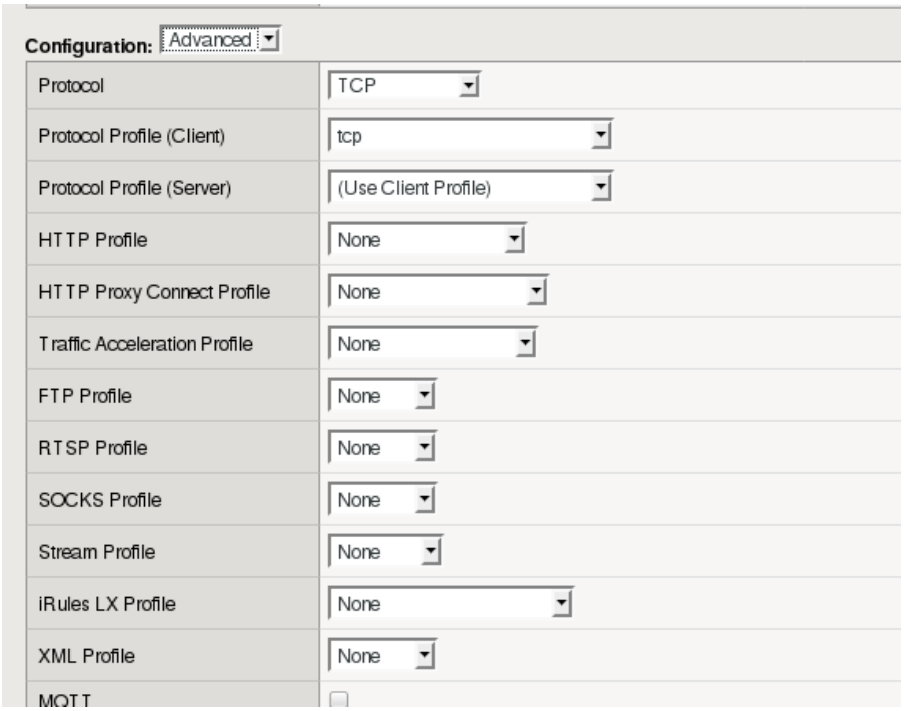
Since iRules LX Streaming does not require the use of TCL iRules, we need a method to associate an LX Plugin to a virtual server. That is done with an iRules LX profile.

1. To create a new iRules LX profile, navigate to the menu **Local Traffic > Profiles > Other > iRules LX** and click the + sign.
2. Name the new profile **ilxlab4_stream_profile**, select the **ilxlab4_stream_pl** LX Plugin and click finish to save the changes.

Assign the iRules LX Profile to Virtual Servers

Now we need to attach our profile to a virtual server.

1. Go into the virtual server **ilxlab4_stream_vs** main configuration **"properties"** window (not the resources tab), then expand the Configuration menu to the advanced setting and you will see the iRules LX Profile setting as shown here:



2. Select the **ilxlab4_stream _profile** then click update at the bottom to save the changes.

Test the ILX Streaming Plugin

Now we should be able to see the hexdumps in the log file. First, in an SSH session with the BIG-IP, tail the log file of the plugin with the following command:

```
# tail -f /var/log/ilx/Common/ilxlab4_stream_pl.hexdump
```

1. Then refresh the page in the browser (URL http://10.0.0.23/ilxlab4stream) and you should see output like this in the SSH terminal:

```
May 10 15:09:33 pid[16974] 00000000: 4745 5420 2f20 4854 5450 2f31 2e31 0d0a  GET./.HTTP/1.1..
May 10 15:09:33 pid[16974] 00000010: 486f 7374 3a20 3130 2e30 2e30 2e32 300d  Host:.10.0.0.20.
May 10 15:09:33 pid[16974] 00000020: 0a55 7365 722d 4167 656e 743a 204d 6f7a  .User-Agent:.Moz
May 10 15:09:33 pid[16974] 00000030: 696c 6c61 2f35 2e30 2028 5831 313b 204c  illa/5.0.(X11;.L
May 10 15:09:33 pid[16974] 00000040: 696e 7578 2078 3836 5f36 343b 2072 763a  inux.x86_64;.rv:
May 10 15:09:33 pid[16974] 00000050: 3532 2e30 2920 4765 636b 6f2f 3230 3130  52.0).Gecko/2010
May 10 15:09:33 pid[16974] 00000060: 3031 3031 2046 6972 6566 6f78 2f35 322e  0101.Firefox/52.
May 10 15:09:33 pid[16974] 00000070: 300d 0a41 6363 6570 743a 2074 6578 742f  0..Accept:.text/
May 10 15:09:33 pid[16974] 00000080: 6874 6d6c 2c61 7070 6c69 6361 7469 6f6e  html,application
May 10 15:09:33 pid[16974] 00000090: 2f78 6874 6d6c 2b78 6d6c 2c61 7070 6c69  /xhtml+xml,appli
May 10 15:09:33 pid[16974] 000000a0: 6361 7469 6f6e 2f78 6d6c 3b71 3d30 2e39  cation/xml;q=0.9
May 10 15:09:33 pid[16974] 000000b0: 2c2a 2f2a 3b71 3d30 2e38 0d0a 4163 6365  ,*/*;q=0.8..Acce
May 10 15:09:33 pid[16974] 000000c0: 7074 2d4c 616e 6775 6167 653a 2065 6e2d  pt-Language:.en-
May 10 15:09:33 pid[16974] 000000d0: 5553 2c65 6e3b 713d 302e 350d 0a41 6363  US,en;q=0.5..Acc
May 10 15:09:33 pid[16974] 000000e0: 6570 742d 456e 636f 6469 6e67 3a20 677a  ept-Encoding:.gz
May 10 15:09:33 pid[16974] 000000f0: 6970 2c20 6465 666c 6174 650d 0a43 6f6e  ip,.deflate..Con
May 10 15:09:33 pid[16974] 00000100: 6e65 6374 696f 6e3a 206b 6565 702d 616c  nection:.keep-al
May 10 15:09:33 pid[16974] 00000110: 6976 650d 0a55 7067 7261 6465 2d49 6e73  ive..Upgrade-Ins
May 10 15:09:33 pid[16974] 00000120: 6563 7572 652d 5265 7175 6573 7473 3a20  ecure-Requests:.
May 10 15:09:33 pid[16974] 00000130: 310d 0a49 662d 4d6f 6469 6669 6564 2d53  1..If-Modified-S
May 10 15:09:33 pid[16974] 00000140: 696e 6365 3a20 4d6f 6e2c 2030 3820 4d61  ince:.Mon,.08.Ma
May 10 15:09:33 pid[16974] 00000150: 7920 3230 3137 2032 323a 3234 3a30 3020  y.2017.22:24:00.
May 10 15:09:33 pid[16974] 00000160: 474d 540d 0a49 662d 4e6f 6e65 2d4d 6174  GMT..If-None-Mat
May 10 15:09:33 pid[16974] 00000170: 6368 3a20 572f 2235 3931 3066 3030 302d  ch:.W/"5910f000-
May 10 15:09:33 pid[16974] 00000180: 3236 3222 0d0a 4361 6368 652d 436f 6e74  262"..Cache-Cont
May 10 15:09:33 pid[16974] 00000190: 726f 6c3a 206d 6178 2d61 6765 3d30 0d0a  rol:.max-age=0..
May 10 15:09:33 pid[16974] 000001a0: 0d0a                                      ..
```

4.1.2 Create and Implement an HTTP server LX Plugin

In this lab exercise, we will use the LX plugin as an HTTP server. The virtual server that we will use this LX Plugin is the **ilxlab4_http_vs** (10.0.0.24) virtual server which does not have a pool attached to it. This VS does not have an HTTP profile associated with it as use of the iRules LX HTTP server requires this configuration.

Review the LX Workspace

1. Go to the LX workspace named **ilxlab4_http**, click on the extension folder named **http_server** and click on the **index.js** file. You should see code that looks like this:

```
 'use strict';
// Use iRules LX as simple HTTP server
var f5 = require('f5-nodejs');

// Create the HTTP request callback function
function httpRequestCallback(req, res) {
  var msg =  '<html><body><h1>ILX HTTP Server</h1>';
  msg += '<p>Welcome client "' + req.client.remoteAddress + '". ';
  msg += 'Your HTTP method is ' + req.method + '.</p>';
  msg += '</body></html>';
  // Set HTTP respond, send reply and close connection.
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end(msg);
}

var plugin = new f5.ILXPlugin();
plugin.startHttpServer(httpRequestCallback);
```

Create the LX Plugin, Profile and Attach to Virtual Server

With our code already in a workspace, all we need to do is create our LX Plugin and iRules LX profile, and attach the profile to the virtual server.

1. Name your LX Plugin **ilxlab4_http_pl**. Create the iRules LX profile with the name of **ilxlab4_http_profile** and attach it to the **ilxlab4_http_vs** virtual server.

Test the ILX HTTP Plugin

1. In your web browser's 2nd tab type in the URL http://10.0.0.24. You should see a web page like this –