BIG-IP® iRulesLX User Guide

Version 13.0

Table of Contents

iRulesLX programmability

What is fundamentally different about iRulesLX? iRulesLX takes advantage of the capabilities of Node.js to enhance the data plane programmability of a BIG-IP® system. To enhance the programmability aspects of iRules®, iRulesLX adds a mechanism to invoke programs in Node.js. Node.js is well-suited to a BIG-IP system, providing a single-threaded environment in which to run programs, taking advantage of asynchronous behavior, and offering a wide range of packages for download. As a developer, the resources of a vast community can help you add functionality to your Node.js applications, while reducing the development effort.

BIG-IP® systems offer an ILX interface that is similar to RPC, as well as a streaming data interface for Node.js. In the case of the former, the ILX interface lets you call a block of code from a TCL iRule to perform a programmatic function in Node.js, such as writing to a database. This guide refers to this technology as ILX RPC. By using the streaming data interface, you can receive and modify traffic in a virtual server. TCP, SSL, HTTP Compression and Web Acceleration profiles, or a combination of the protocols, are supported by the ILX profile included with the streaming data interface. This guide refers to this technology as iRulesLX streaming data. The two interfaces address distinct and different needs but you can develop a plug-in that uses both interfaces.

Why Node.js?

What benefits will you realize with Node.js as a development platform in the next generation of iRules®? The most obvious benefit is that Node.js is written in JavaScript, which is a popular and well-known language among web application developers. For an application developer, not having to learn a new language is one less hurdle to clear should you decide to develop for the iRulesLX platform. Node.js also offers a couple of features that make it suitable for the F5® TMOS® platform.

Node.js runs your code in a single-threaded process. The asynchronous behavior of Node.js improves the runtime performance of server-side JavaScript. If your code depends on the completion of a task like I/O, the runtime places the task in a queue and continues processing your code. Keep in mind that a Node.js process will block in certain circumstances, but Node.js handles I/O requests efficiently. When the I/O task completes, a callback function runs on the results of the task, providing the second noteworthy feature of Node.js. Also of note, Node.js also provides access to binary data.

Node.js runs callback functions upon receiving an event notification, such as the completion of the I/O task mentioned previously that resulted in an event being emitted. In your code, you provide a callback function as a parameter to a function. Because JavaScript supports first-class functions, you can pass a callback as a parameter, and the Node.js runtime will run that function on completion of the I/O task.

By using Node.js, you can enhance iRules functionality and incorporate additional features, such as the use of relational or document databases. While Node.js may not be the best solution in all situations, Node.js offers reasonably high performance capabilities to JavaScript applications, enabling new functionality in iRules.

About Node.js development

iRulesLX adds functionality that enables you to make a call to a Node.js process, run JavaScript code in the Node.js process, and then return the results to ILX RPC. For most extensions that you create, there are two individual yet related tasks in the development process. The first task is the call that invokes the Node.js extension.

The TCL sample code specifies a `when` command and a block of code to run when a specific event occurs. You can call any supported iRules® event, and this sample shows an `HTTP_REQUEST` event on a BIG-IP® system. Within the curly braces, the first line of code uses the iRules command `set` to create a variable, which is the handle for the call. In the handle variable, specify the endpoint and the name of the extension that runs in the Node.js process. The second line of the sample uses the `set` command to create a variable that holds the output of the call.

```
# Sample code for any iRules event

when HTTP_REQUEST {
    # Typical iRule / TCL code
    set hndl [ILX::init "/Common/isc" "Ext1"]
    set result [ILX::call $hndl func arg]
    # Process the result
}
```

The second task to complete for an iRulesLX RPC extension is the Node.js code itself. The first line of the JavaScript sample assigns the `f5-nodejs` package to a variable. If you need other packages in your extension, you should use the `require` method to load them. The second line of the sample uses the `f5` variable to instantiate an instance of an `ILXServer`. The constructor method creates an ILX server object to listen on a port for an event. In this case, `ilx.listen` listens for an event that is generated when a rule invokes `ILX::call`. The callback function takes the request object and tries to locate a function that matches the function named in the argument string. If a match is found, the callback runs the function and returns the results to the caller.

```
/* Load npm or other custom package */

/* Load the f5-nodejs package */
var f5 = require('f5-nodejs');
var ilx = f5.ILXServer();

/* Listen for calls from iRules */

ilx.addMethod('<function name>', function(req, res)
 {
    /* ... typical JavaScript code ... */
    /* Reply with results */
    res.reply(ret); });
ilx.listen();
```

In the event block in the JavaScript code, you can write code to parse the contents of a packet, connect to other services or databases, or cache data. The `res.reply` statement in the JavaScript code returns the results to the `result` variable in the TCL code block.

About packages and modules

Third-party packages and libraries extend the functionality of Node.js. The node package manager (npm) site lists thousands of packages that you can install and use in Node.js extensions. Common packages that you may want to use in iRules® include parsers for JSON and XML, libraries to consume other services and databases, and distributed memory object caching systems like memcached. In addition to frameworks designed to simplify Node.js application development, packages are available for, among others:

- JSON parsers
- XML parsers
- Memcached
- Redis
- MongoDB
- MySQL

Note:

The version of Node.js in the BIG-IP® system offers full Node.js compatibility and supports the same packages as the version of Node.js that you download from the web site.

About tmsh and the iRulesLX environment

The iRulesLX development environment consists of workspaces, extensions, and rules. To simplify the task of creating workspaces and other directories, iRulesLX includes a set of `tmsh` commands to accomplish many of the tasks related to the creation and maintenance of a development environment. If you want to create a simple workspace with a single extension and a single rule, and then publish the rule and attach it to a virtual server, iRulesLX supports that task through a concise set of `tmsh` commands.

iRulesLX follows the Node.js model to take advantage of the common tools that support Node.js. When you edit a workspace extension directory as part of a development environment, the `tmsh` command creates a `package.json` file in the extension directory. The `package.json` file contains the meta data for a Node.js application, and the file also makes the application a valid node package manager (npm) module. The package manager for Node.js (npm) can use `package.json` to install the application on other BIG-IP® systems.

iRulesLX makes use of staging directories for iRules®. Because of this difference, if you edit and then publish an existing rule, you must follow a different procedure for iRulesLX. In this case, you edit the rule in the workspace, not in a production environment.

Tip: *Setting up an environment applies to both ILX RPC and iRulesLX streaming data.*

About the iRulesLX development environment

The development environment for iRulesLX exists in a conventional directory structure (`/var/ilx/workspaces`). Within the directory structure, individual workspaces are identified by a partition, such as `ltm`, as well as a workspace name. For example, a particular workspace may exist in the following path: `/var/ilx/workspaces/partition/workspace name`. The complete directory structure for iRulesLX includes the following directories:

- /var/*ilx*/workspaces/*partition*/*workspace name*
- /var/*ilx*/workspaces/*partition*/*workspace name*/extensions
- /var/*ilx*/workspaces/*partition*/*workspace name*/rules

You can use `ssh` to open a shell in the workspace, or use `tmsh` commands to publish a rule and its associated extensions and packages. The rule and its associated extensions and packages are referred to collectively as a *plug-in*. When a plug-in is created from a workspace, the BIG-IP® system copies the workspace files to a system location. The plug-in runs from that system location.

Working in the iRulesLX development environment

As an iRules® developer, you must create a development environment before you can edit and publish a rule using iRulesLX. Complete these steps to create a workspace, publish a rule from the workspace, and attach the rule to a virtual server.

1. From a `tmsh` command prompt, run the following command to create a workspace.

```
create ilx workspace w
```

2. Using the same `tmsh` command prompt, run the following command to create a rule in the workspace for ILX RPC. The Tcl iRule is not needed for the streaming API.

```
edit ilx workspace w rule r
```

3. To edit an extension in the workspace, run the following `tmsh` command .

```
edit ilx workspace w extension e
```

4. To edit a file in the extension directory, run the following `tmsh` command.

```
edit ilx workspace w extension e file f
```

5. To create a plug-in from the developer workspace, run the following `tmsh` command.

```
create ilx plugin p from-workspace w
```

6. To attach the rule to a virtual server, run the following `tmsh` command.

```
modify virtual v rule w/r
```

You have now created a development environment, a plug-in, and attached the rule to a virtual server.

Republishing in the iRulesLX environment

As an iRules® developer, you must edit a rule in the development environment before you can republish a rule using iRulesLX. Complete the following steps to republish a rule from a workspace.

1. Using a `tmsh` command prompt, run the following command to edit an existing rule in the workspace.

```
edit ilx workspace w rule r
```

2. To edit a file in the extension directory, run the following `tmsh` command.

```
edit ilx workspace w extension e file f
```

   You may use this particular command as often as necessary to edit files in the extension directory.

3. To republish the modified plug-in, run the following command.

```
modify ilx plugin p from-workspace w
```

You have now modified and republished the plug-in.

About iRulesLX graphical editor

iRulesLX provides tools to edit Node.js extensions and manage plug-ins. Management tasks include creating, importing, and exporting workspaces, as well as enabling and disabling plug-ins, or modifying the properties of a plug-in. Editing features include the ability to open a file by double-clicking a file in the workspace, line numbering, syntax highlighting, and matching of parentheses and braces within a file. Using the graphical editor, you can produce a plug-in from a workspace and enable or disable a plug-in to run on a BIG-IP® system.

Tip: *Using the graphical editor applies to both ILX RPC and iRulesLX streaming data.*

Creating a new workspace

Manage an iRulesLX workspace by using the Traffic Management User Interface (TMUI) to access a BIG-IP® system.

1. Log in to the BIG-IP system with your user name and password.
2. On the Main tab, click **iRules**.
3. Click **LX Workspaces** to display the list of existing iRulesLX workspaces.
   You must provision ILX (`System\Resource Provisioning`) in order to view the ILX menu items.
4. Click the **Create** button.
5. When prompted, type a name for the new workspace.

Deleting a workspace

You can reduce clutter by deleting workspaces that you no longer need or use.

1. On the Main tab, click **iRules**.
2. Click **LX Workspaces** to display the existing workspaces.
3. Select a workspace to delete.
4. Click the **Delete** button.
   When you delete a workspace, you delete the contents of the workspace, as well as the workspace.

Exporting a workspace

To save time and work, you can export a workspace to another BIG-IP® system.

1. On the Main tab, click **iRules**.
2. Click **LX Workspaces** to display the existing workspaces.
3. Select a workspace to export.

4. Click the **Export** button.
   You can also use the export functionality to archive a workspace.

Importing a workspace

To leverage existing iRules®, you can import a workspace from another BIG-IP® system.

1. On the Main tab, click **iRules**.
2. Click **LX Workspaces** to display the existing workspaces.
3. Click the **Import** button.
4. Select a workspace to import.
   When you import a workspace, you must choose the source of the workspace, such as the name of an archive file, a URI that identifies an archive, a workspace, or a plug-in.

Adding a rule in the workspace editor

For individual workspaces, you can use the workspace editor to create a new rule or make changes to an existing rule.

1. Click **Add iRule** to add a rule.
   The workspace editor screen appears when you create or import a workspace, or when you open a workspace to make modifications.
2. To delete any rule, extension, or extension file, select the item and click the **Delete** button.

Adding an extension in the workspace editor

For individual workspaces, you can use the workspace editor to create a new extension or make changes to an existing extension.

1. Click the **Add Extension** button to add an extension to a workspace.
   In this context, an extension consists of scripts, files, or Node.js modules.
2. To delete any rule, extension, or extension file, select the item and click the **Delete** button.

Adding a file to an extension in the workspace editor

When you are working with an extension in the workspace, you can use the workspace editor to add a file to an existing extension.

1. Click the **Add Extension File** button to add a file to an extension.
   You must select an extension to enable the button.
2. To delete any rule, extension, or extension file, select the item and click the **Delete** button.

Reverting to a previous version in the workspace editor

As a convenience, you can revert any unsaved changes to a file and restore the previous version.

1. Click the **Revert File** button.

   If you make a number of changes and then decide not to continue, you can restore the previous version of a file that is open in the editing panel. To undo an individual change to a file, use the Ctrl+Z key combination.

2. To save the changes, rather than revert to the previous saved copy of a file, click the **Save File** button.

   When you save the changes, you are saving the changes to the file open in the editing pane.

Viewing plug-in properties

You can click on a plug-in to view its properties using the LX Plugins screen settings.

1. To reload a plug-in from a workspace, click **Reload from Workspace**.

   You can select a workspace other than the workspace used to originally create the plug-in. The list of available workspaces appears in the drop down list. When you choose to reload the workspace, you incorporate workspace changes into the plug-in. Reloading a workspace integrates any changes you made to a workspace that are not yet part of a plug-in. By creating multiple workspaces, you can implement multiple versions of a plug-in.

2. To view the extensions properties, click on one of the extensions listed in **Extensions**.

Viewing extension properties

You can access any of the available property settings for an extension and make a change, such as entering a value or selecting a value from a drop-down list.

1. To specify a concurrency mode for the extension, select a value from the drop-down list.

   The **Dedicated** setting specifies a separate Node.js process for each provisioned Traffic Management Microkernel (TMM). **Single** specifies a single Node.js process for all TMM processes.

2. To specify the maximum number of restarts for an extension, type a value in the **Maximum Restarts** field.

   Specifies the maximum failures for an extension process before the system abandons efforts to restart the process. The default value is 5.

3. To specify a time interval for maximum restarts, type a value in the **Restart Interval** field.

   Specifies the time, in seconds, that the maximum number of restarts (Maximum Restarts) can occur. The default value is 60 seconds.

4. To enable debugging, check the **EnableDebug** setting.

   Enable or disable debug mode for the extension. You must restart the plugin for the setting to take effect.

5. To specify a range of port numbers, type a value for the **Debug Port Range Low** and **Debug Port Range High** fields.

After you enable debugging, the iRulesLX process searches for an available port to attach the node inspector. The low value represents the low end of the port range that iRulesLX will try, and the high value represents the high end of the port range. iRulesLX starts with the low end port number and increments the value until it locates an available port or reaches the high end port.

Creating an iRulesLX plug-in

After you save the workspace files, create a plug-in from the workspace by navigating to the LX Plugins screen and following the steps to create a plug-in.

1. On the Main tab, select **LX Plugins**, and click the **Create** button.
2. For the new plug-in, type a name for the plug-in and select the corresponding workspace for the plug-in from the drop-down list. You can provide a description for the plug-in you are creating, although the description is optional.
3. Click **Finished** to create the plug-in.

   Click **Repeat** to create the plug-in if you want to create a similar plug-in with a different name. The repeat feature uses the same settings.

Once you have created a plug-in, you can begin using it by attaching the corresponding rule to a virtual server.

iRulesLX streaming data

While IPX RPC addresses a need for utility applications, such as writing data to a database, iRulesLX focuses on receipt and modification of network traffic, including protocols not currently supported by a BIG-IP system. This functionality represents the primary purpose of iRulesLX streaming data. The ILX profile included with iRulesLX works with any combination of TCP, SSL, HTTP compression and Web Acceleration profiles available on a BIG-IP system. By using the iRulesLX streaming data API, you can create a plug-in to manage unsupported protocols or add custom behavior to supported protocols. To make iRulesLX plug-in programming as simple and powerful as possible, the API consists of methods that you can call from a Node.js application. An iRulesLX plug-in can contain Node.js code that you write, contributed modules and libraries, or both.

The iRulesLX streaming data API includes methods to perform the operations to receive and modify data and to manage the flow of data between a client and a server. iRulesLX streaming provides notification of events, such as client connections to a virtual server, server connections, disconnections, access to data groups, access to the session DB, as well as enhancements to the Node.js buffer methods. To assist with troubleshooting and performance, iRulesLX streaming data API includes methods for debugging and tracing a plug-in.

To maintain consistency, the workspace setup and configuration of iRulesLX streaming is similar to the setup and configuration for ILX RPC. The iRulesLX (ILX) streaming data flows and ILX RPC share workspace and plug-in configuration to simplify setup and deployment activities. Likewise, starting a plug-in, stopping a plug-in, and restarting a plug-in after you modify it are consistent for all plug-ins. ILX streaming does not require a Tcl iRule to be associated with a virtual server.

The iRulesLX streaming data objects and methods are described in the following topics.

ILXPlugin class

The ILXPlugin class provides configuration for the plug-in interface and responds to new client connections to a virtual server.

ILXPlugin.start

ILXPlugin.start initializes communication with the TMM. For more information, see the ILXPluginOptions reference.

```
ILXPlugin.start ( ILXPluginOptions )
```

ILXPlugin.on

ILXPlugin.on emits the `initialized` event when the plug-in successfully connects to the TMM. A virtual server with an ILX profile that refers to the plug-in is necessary for the event to occur. The event signals that the plug-in may issue ILXDatagroup, ILXTable, and ILXStream.connect commands.

```
ILXPlugin.on ('initialized', function () {...})
```

ILXPlugin.on emits the `uninitialized` event when the plug-in is no longer connected to the TMM. In contrast to the `initialized` event, the association between a virtual server with an ILX profile and a

plug-in no longer exists when the event occurs. The event signals the plug-in that it cannot issue ILXDatagroup, ILXTable, and ILXStream.connect commands.

```
ILXPlugin.on ('uninitialized', function () {...})
```

ILXPlugin.on listens for a connection request and invokes the callback function when the event occurs. The flow parameter shown in the example is an ILXFlow object that contains the ILXFlow.client and ILXFlow.server socket streams.

```
ILXPlugin.on ( 'connect', function (flow) { ... } )
```

You can use ILXPlugin.on as shown:

```
var f5 = require("f5-nodejs");
var plugin = new f5.ILXPlugin();
plugin.on("connect", function(flow)) {
...
}
```

ILXPlugin.setGlobalTraceLevel

ILXPlugin.setGlobalTraceLevel enables or disables debug and tracing output for the application plug-in, including flows and streams.

```
ILXPlugin.setGlobalTraceLevel ( integer )
```

ILXPlugin.globalTraceLevel

ILXPlugin.globalTraceLevel returns the current global trace level.

```
ILXPlugin.globalTraceLevel()
```

ILXPlugin.setTraceLevel

ILXPlugin.setTraceLevel enables or disables debug and tracing for the ILXPlugin object.

```
ILXPlugin.setTraceLevel ( integer )
```

ILXPlugin.traceLevel

ILXPlugin.traceLevel returns the current trace level.

```
ILXPlugin.traceLevel()
```

ILXPlugin.getDataGroup

ILXPlugin.getDataGroup returns an ILXDatagroup object, which defines a set of operations that delegate work to the C++ implementation.

```
ILXPlugin.getDataGroup (dg_name)
```

ILXPluginOptions class

The ILXPluginOptions class defines actions taken on parameters passed to `ILXPlugin.start`.

ILXPluginOptions.handleClientOpen

If `true`, ILXPluginOptions.handleClientOpen requests that the plug-in perform validation before permitting the connection request to proceed, or terminating the client connection and flow.

```
ILXPluginOptions.handleClientOpen ( Boolean )
```

ILXPluginOptions.handleClientData

If `true`, indicates that the plug-in expects to receive payload data from the client stream ILXFlow.client.

```
ILXPluginOptions.handleClientData ( Boolean )
```

ILXPluginOptions.handleServerData

If `true`, indicates that the plug-in expects to receive payload data from the server stream ILXFlow.server.

```
ILXPluginOptions.handleServerData ( Boolean )
```

ILXPluginOptions.disableServer

Indicates that the ILXFlow.server socket is disabled. The default value is `false`. You may use this if a plug-in only interacts with the client, or the plug-in is invoked with ILXPlugin.startHttpServer. Note that calling ILXPlugin.startHttpServer to invoke a plug-in automatically enables the setting.

```
ILXPluginOptions.disableServer( Boolean )
```

IXFlow class

The ILXFlow class manages operations like closing a stream, detaching from the TMM, or making a load balancing choice.

ILXFlow.lbSelect

ILXlbOptions specifies the load balancing options to invoke. Data may not be sent to the server prior to calling ILXFlow.lbSelect, and you must enable `ILXPluginOptions.handleClientOpen`.

```
ILXFlow.lbSelect ( ILXlbOptions )
```

ILXFlow.end

Results in a graceful shutdown of client and server streams.

```
ILXFLow.end()
```

ILXFlow.destroy

Results in an immediate shutdown of client and server streams.

```
ILXFlow.destroy()
```

ILXFlow.detach

Detaches the plug-in from the Traffic Management Microkernel (TMM). The client-server connection remains active in TMM, but data and events are not passed to the plug-in.

```
ILXFLow.detach()
```

ILXFlow.client

Specifies the client ILXStream socket object.

ILXFlow.server

Specifies the server ILXStream socket object.

ILXFlow.virtual

Specifies the virtual server associated with the client side flow. The object contains the following properties:

- ILXFLow.virtual.name
- ILXFlow.virtual.address
- ILXFlow.virtual.port
- ILXFlow.virtual.routeDomain

ILXFlow.lb

Specifies the information about the load balancing selection made by, or for, the flow.

- ILXFlow.lb.virtualServer, which is undefined unless the connection was load balanced to a virtual server.
- ILXFlow.lb.pool, which is undefined if ILXFlow.lb.virtualServer is defined.
- ILXFlow.lb.remote, consisting of the remote server address, ILXFLow.lb.remote.address, ILXFLow.lb.remote.port, or ILXFlow.lb.remote.routeDomain.
- ILXFlow.lb.vlan, which is a numeric VLAN ID. In the case of a virtual server targeting a virtual server, the value will be zero.
- ILXFLow.lb.nexthop, which is undefined if the `nexthop` is all zeroes.

  If SNAT is enabled, the following fields will be populated:

  - ILXFLow.lb.snatpool
  - ILXFlow.lb.local.address
  - ILXFlow.lb.local.port
  - ILXFlow.lb.local.routeDomain

ILXFLow.on

The event is emitted when a flow level error occurs. The flow becomes unusable.

```
ILXFlow.on('error', function() {...})
```

The event is emitted when both client and server streams have closed.

```
ILXFLow.on('close', function() {...})
```

ILXFlow.setTraceLevel

Enables or disables debug tracing of an ILXFLow object.

```
ILXFLow.setTraceLevel( integer )
```

ILXFlow.traceLevel

Determines the current trace level.

```
ILXFLow.traceLevel()
```

ILXFlow.tmmId

Determines the identifier of the TMM that is supporting the flow, as a positive integer.

```
ILXFlow.tmmId()
```

ILXStream class

The ILXStream class implements the Node.js stream and socket interfaces stream.readable, stream.Writeable, stream.Duplex, and net.Socket.

A client stream and a server stream are created for each ILXFlow object instance. ILXStream also defines a method that a plug-in uses to initiate outbound connections that go directly to the TMM by using the V1 plug-in interface.

ILXStream.allow

The ILXStream.allow method allows a client connection to proceed to a server. You must call this method from an ILXFlow.client stream. `ILXPluginOptions.handleClientOpen` must be enabled and no data may have been sent to the server prior to calling ILXStream.allow.

```
ILXStream.allow()
```

ILXStream.on

The ILXStream.on method emits the connect event when an ILXFlow.server stream connects to a server. The event is emitted only for the ILXFLow.server object.

```
ILXStream.on('connect', function () {...})
```

The ILXStream.on method emits the `requestStart` event after the TMM receives HTTP request headers. An HTTP profile must be associated with the virtual server.

```
ILXStream.on('requestStart', function (uri, method, path, query, version,
hdrs) {...})
```

The ILXStream.on method emits the `requestComplete` event after the TMM receives a complete HTTP request transaction from the client. An HTTP profile must be associated with the virtual server.

```
ILXStream.on('requestComplete', function () {...})
```

The ILXStream.on method emits the `responseStart` event after the TMM receives HTTP response headers. An HTTP profile must be associated with the virtual server.

```
ILXStream.on('responseStart', function (hdrs, status, version) {...})
```

The ILXStream.on method emits the `responseComplete` event after the TMM receives a complete HTTP response transaction from the server. An HTTP profile must be associated with the virtual server.

```
ILXStream.on('responseComplete', function () {...})
```

ILXStream.setTraceLevel

Specifies the level of debug/tracing of an ILXStream object.

```
ILXStream.setTraceLevel( integer )
```

ILXStream.traceLevel

Retrieves the current trace level setting.

```
ILXStream.traceLevel()
```

ILXStream.tmmId

Determines the identifier of the TMM that is supporting the stream, as a positive integer.

```
ILXStream.tmmId()
```

ILXStream.connect

Specifies the options for a connection.

Property	Description
options.host	Specifies an IP address. The address may contain a route domain qualifier, such as 10.10.0.1%5 to specify an address in route domain 5.
options.port	Specifies an IP port.
options.virtualServer	Specifies a virtual server. A connection will use the server-side stack of the virtual server. This setting may not be used in conjunction with the host and port (host:port) options.
options.virtualProtocolStack	Specifies the server-side profile of a specific virtual server when making a connection. The virtual server must be associated with the current plug-in by an ILX profile. Note that a `virtualProtocolStack` is independent of the `virtualServer` option. A connection cannot be established if the plug-in is not associated with any virtual server specified in an ILX profile. If you do not specify a virtual server, a virtual server is chosen.
options.disableSsl	If an SSL profile is present on the server side of the virtual server, use this option to disable SSL for a plug-in initiated connection.

```
ILXStream.connect( options )
```

ILXBufferUtil class

The ILXBufferUtil class defines methods that manipulate inbound and outbound plug-in data.

ILXBufferUtil.append

The ILXBufferUtil.append method appends `bytes`, either as a string or a buffer of data, to a specified `buffer`.

```
ILXBufferUtil.append( buffer, bytes )
```

ILXBufferUtil.erase

The ILXBufferUtil.erase method removes bytes from a specified `buffer`, starting at an `offset`, and removing `len` bytes from the buffer.

```
ILXBufferUtil.erase( buffer, offset, len )
```

ILXBufferUtil.insert

The ILXBufferUtil.insert method inserts the specified `bytes` into a `buffer`, starting at an `offset`.

```
ILXBufferUtil.insert( buffer, bytes, offset )
```

ILXBufferUtil.replace

The ILXBufferUtil.replace method replaces the first instance of a string in a `buffer`. The string to replace is specified by `old`, and the replacement string is specified by `repl`. The `result` object holds the output of the method and the indices of the words replaced by the method. If no matches were found, the array remains empty and the original buffer is returned.

Property	Description
options.offset	Specifies the starting location in the buffer. Assumes the index to the buffer starts at zero.
options.icase	Indicates whether to ignore case when searching; false, by default.
options.all	Indicates whether to replace all occurrences of the string, or just the first occurrence; false, by default.

```
ILXBufferUtil.replace( buffer, old, repl, result, options )
```

ILXBufferUtil.replaceAt

The ILXBufferUtil.replaceAt method creates a new `buffer` that contains one or more replaced tokens. The bytes starting with `offset`, up to the specified length `len` are replaced with the bytes specified by `repl`.

Property	Description
options.length	Specifies the number of bytes to replace. Default length is the end of the buffer.

```
ILXBufferUtil.replaceAt( buffer, repl, offset, len )
```

ILXBufferUtil.search

The ILXBufferUtil.search method finds the index of the specified `bytes` in a `buffer`. If the bytes are not found in the string, the index is set to -1.

```
ILXBufferUtil.search( buffer, bytes, options )
```

ILXBufferUtil.rsearch

The ILXBufferUtil.rsearch method finds the index of the specified bytes in a buffer from the end of the string.

Property	Description
options.offset	Specifies the starting location in the buffer. Assumes the index to the buffer starts at zero.
options.icase	Indicates whether to ignore case when searching; false, by default.

ILXDatagroup class

The ILXDatagroup class provides an API data group access, similar in functionality to an iRules® data group.

ILXDatagroup.getSize

The ILXDatagroup.getSize method retrieves the number of elements in a data group.

```
ILXDatagroup.getSize()
```

ILXDatagroup.getType

The ILXDatagroup.getType method retrieves the type of the data group. Possible return values are:

- Type.IP = 1
- Type.STRING = 2
- Type.INTEGER = 3

```
ILXDatagroup.getType()
```

ILXDatagroup.matchEquals

The ILXDatagroup.matchEquals method searches a data group with a key and retrieves values based on the return type. The parameter name specifies the name of the record as a key to match in the data group, and is dependent on the type of the data group (string, number, or IP). An IP address argument can be IPv4 or IPv6 values, such as 10.0.0.0/24, 2001:db8::1/64, or ::/32.

Property	Description
options.return	Specifies that only the found name is retrieved.
options.all	Specifies that all found matches are retrieved.
options.icase	Indicates a non case-sensitive search, if true. Defaults to false.

The options.return property accepts the following values:

Property	Description
ILXDatagroup.Options.NAME	Specifies that just the name is included in the results.
ILXDatagroup.Options.VALUE	Specifies that the value corresponding to the name is included in the results.
ILXDatagroup.Options.ELEMENT	Specifies that the object that includes the name and value is included in the results.
none	Indicates that the Boolean value is included in the results; true if a match is found, otherwise false.

ILXDatagroup.searchEquals

Refer to the description for ILXDatagroup.matchEquals.

```
ILXDatagroup.searchEquals( name, options )
```

ILXDatagroup.searchStartsWith

The ILXDatagroup.searchStartsWith method retrieves records from a data group where the name matches the prefix and the type matches the options type. Valid only for data groups of type Type.STRING.

Property	Description
options.return	Specifies that only the found name is retrieved.
options.all	Specifies that all found matches are retrieved.
options.icase	Indicates that a non case-sensitive search, if true. Defaults to `false`.

The options.return property accepts the following values:

Property	Description
ILXDatagroup.Options.NAME	Specifies that just the name is included in the results.
ILXDatagroup.Options.VALUE	Specifies that the value corresponding to the name is included in the results.
ILXDatagroup.Options.ELEMENT	Specifies that the object that includes the name and value is included in the results.
none	Indicates that the Boolean value is included in the results; `true` if a match is found, otherwise `false`.

```
ILXDatagroup.searchStartsWith( name_prefix, options )
```

ILXDatagroup.searchEndsWith

The ILXDatagroup.searchEndsWith method retrieves records from a data group where the name matches the suffix and the type matches the options type. Valid only for data groups of type Type.STRING.

Property	Description
options.return	Specifies that only the found name is retrieved.
options.all	Specifies that all found matches are retrieved.
options.icase	Indicates a non case-sensitive search, if true. Defaults to `false`.

The options.return property accepts the following values:

Property	Description
ILXDatagroup.Options.NAME	Specifies that just the name is included in the results.
ILXDatagroup.Options.VALUE	Specifies that the value corresponding to the name is included in the results.
ILXDatagroup.Options.ELEMENT	Specifies that the object that includes the name and value is included in the results.

Property	Description
none	Indicates that the Boolean value is included in the results; `true` if a match is found, otherwise `false`.

```
ILXDatagroup.searchEndsWith( name_suffix, options )
```

ILXDatagroup.searchContains

The ILXDatagroup.searchContains method retrieves records from a data group where the name includes the token and the type matches the options type. The token parameter specifies the name string to match in a record. Valid only for data groups of type Type.STRING.

Property	Description
options.return	Specifies that only the found name is retrieved.
options.all	Specifies that all found matches are retrieved.
options.icase	Indicates a non case-sensitive search, if true. Defaults to `false`.

The options.return property accepts the following values:

Property	Description
ILXDatagroup.Options.NAME	Specifies that just the name is included in the results.
ILXDatagroup.Options.VALUE	Specifies that the value corresponding to the name is included in the results.
ILXDatagroup.Options.ELEMENT	Specifies that the object that includes the name and value is included in the results.
none	Indicates that the Boolean value is included in the results; `true` if a match is found, otherwise `false`.

```
ILXDatagroup.searchContains( token, options )
```

ILXDatagroup.matchStartsWith

The ILXDatagroup.matchStartsWith method retrieves records that represent prefixes for the name and the type matches the options type. The name parameter specifies the matching string. Valid only for data groups of type Type.STRING.

Property	Description
options.return	Specifies that only the found name is retrieved.
options.all	Specifies that all found matches are retrieved.
options.icase	Indicates a non case-sensitive search, if true. Defaults to `false`.

The options.return property accepts the following values:

Property	Description
ILXDatagroup.Options.NAME	Specifies that just the name is included in the results.

Property	Description
ILXDatagroup.Options.VALUE	Specifies that the value corresponding to the name is included in the results.
ILXDatagroup.Options.ELEMENT	Specifies that the object that includes the name and value is included in the results.
none	Indicates that the Boolean value is included in the results; `true` if a match is found, otherwise `false`.

```
ILXDatagroup.matchStartsWith( name, options )
```

ILXDatagroup.matchEndsWith

The ILXDatagroup.endsWith method retrieves records that represent suffixes for the name and the type matches the options type. The line parameter specifies the matching string. Valid only for data groups of type Type.STRING.

Property	Description
options.return	Specifies that only the found name is retrieved.
options.all	Specifies that all found matches are retrieved.
options.icase	Indicates a non case-sensitive search, if true. Defaults to `false`.

The options.return property accepts the following values:

Property	Description
ILXDatagroup.Options.NAME	Specifies that just the name is included in the results.
ILXDatagroup.Options.VALUE	Specifies that the value corresponding to the name is included in the results.
ILXDatagroup.Options.ELEMENT	Specifies that the object that includes the name and value is included in the results.
none	Indicates the Boolean value is included in the results; `true` if a match is found, otherwise `false`.

```
ILXDatagroup.endsWith( line, options )
```

ILXDatagroup.matchContains

The ILXDatagroup.matchContains method retrieves records contained in the name and the type matches the options type. The name parameter specifies the matching string. Valid only for data groups of type Type.STRING.

Property	Description
options.return	Specifies that only the found name is retrieved.
options.all	Specifies that all found matches are retrieved.
options.icase	Indicates a non case-sensitive search, if true. Defaults to `false`.

The options.return property accepts the following values:

Property	Description
ILXDatagroup.Options.NAME	Specifies that just the name is included in the results.
ILXDatagroup.Options.VALUE	Specifies that the value corresponding to the name is included in the results.
ILXDatagroup.Options.ELEMENT	Specifies that the object that includes the name and value is included in the results.
none	Indicates that the Boolean value is included in the results; `true` if a match is found, otherwise `false`.

```
ILXDatagroup.matchContains( name, options )
```

ILXDatagroup.forEach

The ILXDatagroup.forEach method iterates through the data group and invokes a callback function for every record. The `callback_function` parameter defines a callback as an index and an element, where the index is the index of the current element and the element is the name and value for the record object. An example of a callback function that iterates the first 10 records in a data group is shown here:

```
dg.forEach(function(index, element) {
    if (index >= 10) {
        return true;
    }
    console.log("name: " + element.name + " value: " + element.value);
});
```

ILXLbOptions class

The ILXLbOptions class represents the options available to the ILXFlow.lbSelect method.

ILXLbOptions

Property	Description
ILXLbOptions.virtualserver	Specifies the name of the BIG-IP® LTM virtual server, as a fully-qualified path name.
ILXLbOptions.poolName	Specifies the name of the LTM pool name, as a fully-qualified path name.
ILXLbOptions.remote.address	Specifies the IP address to which to connect. The address may include a route domain, such as 10.1.1.1%22, to specify route domain 22.
ILXLbOptions.remote.port	Specifies the port number to which to connect.
ILXLbOptions.interface	Specifies the physical interface name.
ILXLbOptions.includeVirtualServers	Indicates whether to include virtual servers. Must be set to true to if an address is specified in ILXLbOptions.remote.address.

Load balancing uses the following precedence rules:

- The system will load balance to the specified virtual server.
- The system will load balance to the specified pool.
- The system will load balance to the specified remote address if no virtual server or pool is specified. If ILXLbOptions.includeVirtualServers is `true`, a virtual server may be specified by remote address.

ILXTable class

The ILXTable class defines an asynchronous API to access the TMM session DB.

ILXTable

The ILXTable class defines an asynchronous API to access the TMM session DB. API methods and parameters mimic the Tcl iRules® table commands. Features of the API include the following:

- The session DB provides storage for key-value pairs that you can share across connections and plug-in processes.
- The class lets you create and query key-value pairs.
- Upon completion of an operation, a table emits an event. The Node.js event `complete` includes result and status.
- You access table operations by using the table property of an ILXStream object. ILXStream.table returns an ILXTable object.

```
var myTableRequest = flow.client.table.set("myKey", "value");
myTableRequest.on('set', function (value, status) {...});
```

ILXTable.set

The ILXTable.set method sets a session DB key-value pair, where `key`, `value`, and `options` are parameters to the method. The options parameter to the method offers the following properties:

Property	Description
options.excl	If true, an existing key will not be updated, and the value will be returned. Default value is `false`.
options.lifetime	Specifies the number of seconds, or `ILXTable.INDEFINITE`. Default value is `ILXTable.INDEFINITE`.
options.mustExist	Indicates that a key will be created if it does not exist. Default value is `false`. If the value is `true` and the `key` does not exist, no change occurs.
options.noReply	Indicates that the TMM will not send a reply to the plug-in and a `complete` event will not be generated. Default value is `false`.
options.noTouch	Indicates that the time stamp of the `key` will not be updated. Default value is `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.

Property	Description
options.timeout	Specifies the number of seconds, or `ILXTable.INDEFINITE`. Default value is 180 seconds.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a key value, a system error, or one of the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND
- ILXTable.EXISTS

```
ILXTable.set( key, value, options )
```

ILXTable.add

The ILXTable.add method adds a session DB key-value pair, where `key`, `value`, and `options` are parameters to the method. The ILXTable.set method with the `excl` option set to `true` produces the same result. The `options` parameter to the method offers the following properties:

Property	Description
options.lifetime	Specifies the number of seconds, or `ILXTable.INDEFINITE`. Default value is `ILXTable.INDEFINITE`.
options.noReply	Indicates that the TMM will not send a reply to the plug-in and a `complete` event will not be generated. Default value is `false`.
options.noTouch	Indicates that the timestamp of the key will not be updated. Default value is `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.
options.timeout	Specifies the number of seconds, or `ILXTable.INDEFINITE`. Default value is 180 seconds.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a key value, a system error, or one of the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.add( key, value, options )
```

ILXTable.replace

The ILXTable.replace method updates a session DB key-value pair, where `key`, `value`, and `options` are parameters to the method. The ILXTable.set method with the `mustExist` option set to `true` produces the same result. The `options` parameter to the method offers the following properties:

Property	Description
options.lifetime	Specifies the number of seconds, or `ILXTable.INDEFINITE`. Default value is `ILXTable.INDEFINITE`.
options.noReply	Indicates that the TMM will not send a reply to the plug-in and a `complete` event will not be generated. Default value is `false`.
options.noTouch	Indicates that the timestamp of the key will not be updated. Default value is `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.
options.timeout	Specifies the number of seconds, or `ILXTable.INDEFINITE`. Default value is 180 seconds.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a key value, a system error, or one of the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.replace( key, value, options )
```

ILXTable.lookup

The ILXTable.lookup method retrieves a session DB key-value pair, where `key` and `options` are parameters to the method.

Property	Description
options.noTouch	Indicates that the time stamp of the `key` will not be updated. Default value is `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a key value, a system error, or one of the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.lookup( key, options )
```

ILXTable.incr

The ILXTable.incr method increments a value, where `key` and `options` are parameters to the method.

Property	Description
options.delta	Specifies the increment. Default value is 1.

Property	Description
options.mustExist	Indicates that a `key` will be created if it does not exist. Default value is `false`. If the value is `true` and the `key` does not exist, no change occurs.
options.noReply	Indicates that the TMM will not send a reply to the plug-in and a `complete` event will not be generated. Default value is `false`.
options.noTouch	Indicates that the time stamp of the `key` will not be updated. Default value is `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a key value, a system error, or one of the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.incr( key, value, options )
```

ILXTable.append

The ILXTable.append method appends a string to a session DB value, where `key`, `value`, and `options` are parameters to the method.

Property	Description
options.mustExist	Indicates that a key will be created if it does not exist. Default value is `false`. If the value is `true` and the `key` does not exist, no change occurs.
options.noReply	Indicates that the TMM will not send a reply to the plug-in and a `complete` event will not be generated. Default value is `false`.
options.noTouch	Indicates that the time stamp of the key will not be updated. Default value is `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a key value, a system error, or one of the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.append( key, value, options )
```

ILXTable.delete

The ILXTable.delete method deletes a session DB key and its associated value, where key and options are parameters to the method.

Property	Description
options.noReply	Indicates that the TMM will not send a reply to the plug-in and a `complete` event will not be generated. Default value is `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a system error or one of the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.delete( key, options )
```

ILXTable.deleteAll

The ILXTable.deleteAll method deletes all key-value pairs in a session DB subtable.

Property	Description
options.noReply	Indicates that the TMM will not send a reply to the plug-in and a `complete` event will not be generated. Default value is `false`.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a system error or the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.deleteAll( subtable, options )
```

ILXTable.setTimeout

The ILXTable.setTimeout method sets a timeout value for a session DB key.

Property	Description
options.noReply	Indicates that the TMM will not send a reply to the plug-in and a `complete` event will not be generated. Default value is `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a timeout value, a system error, or one of the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.setTimeout( key, value, options )
```

ILXTable.getTimeout

The ILXTable.getTimeout method retrieves the timeout value for a session DB key.

Property	Description
options.remaining	Indicates that the method return remaining time instead of timeout value. Defaults to `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is `0`.

The method returns a timeout or time remaining value, a system error, or the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.getTimeout( key, options )
```

ILXTable.setLifetime

The ILXTable.setLifetime method sets the lifetime of a key, in seconds, or `ILXTable.INDEFINITE`.

Property	Description
options.noReply	Indicates that the TMM will not send a reply to the plug-in and a `complete` event will not be generated. Default value is `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a lifetime value, a system error, or one of the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.setLifetime( key, value, options )
```

ILXTable.getLifetime

The ILXTable.getLifetime method returns the lifetime value for a session DB key.

Property	Description
options.remaining	Indicates that the method return remaining time instead of timeout value. Defaults to `false`.
options.subtable	Specifies the name of the subtable. Default value is no subtable.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is `0`.

The method returns a lifetime or lifetime remaining value, a system error, or one of the following status values:

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.getLifetime( key, options )
```

ILXTable.keys

The ILXTable.keys method returns the existing keys in a subtable in the session DB.

Property	Description
options.count	Indicates that the method return a count of keys, not the keys and values. Defaults to `false`.
options.noTouch	Indicates that the timestamp of the key will not be updated. Default value is `false`.
options.traceLevel	Specifies the level of debug tracing for an operation. Default value is 0.

The method returns a count of keys or an array of keys, a system error, or one of the following status values

- ILXTable.OK
- ILXTable.NOT_FOUND

```
ILXTable.keys( subtable, options )
```

ILXTransaction class

The ILXStream flow for the client and the server contains ILXTransaction objects. The ILXTransaction objects provide properties and methods for the client and server traffic.

The ILXTransaction class provides the context for management of request and response transactions. The association of ILX and HTTP profiles with a virtual server indicates that traffic management be performed in the context of request and response transactions. A request transaction begins with the following event:

```
ILXFlow.client.on('requestStart', function(request) {...})
```

A response transaction begins with the following event:

```
ILXFlow.server.on('responseStart', function(response) {...})
```

All request and response headers are available at the start of a transaction. The request or response must be read using standard Node.js data or readable events on the ILXFlow.client or ILXFlow.server. For comparison, the end of a transaction is indicated by the events

```
ILXFlow.client.on('requestComplete', function(request) {...})
```

```
ILXFlow.server.on('responseComplete', function(response) {...})
```

At the end of the transaction, the headers and body are available to the plug-in.

ILXTransaction.complete

The ILXTransaction.complete method forwards a request or response. If the ILXTransaction object is part of a request, the method is called to forward a request to the server. If the ILXTransaction object is part of a response, the method is called to forward a response to the client.

```
ILXTransaction.complete()
```

ILXTransaction.respond

The ILXTransaction.respond method called on a request object indicates that a plug-in will respond directly to a client. The request will be discarded and not sent to the server. When the `requestComplete` event is received, a plug-in may call ILXtransaction.setHeader to add the headers to the response. Likewise, a plug-in may add a body to the response by calling ILXFlow.client.write. To send the response to the client, the plug-in must call ILXTransaction.complete.

ILXTransaction.removeHeader

The ILXTransaction.removeHeader method prevents the named header from being returned to a client in a response, or sent to a server in a request.

```
LIXTransaction.removeHeader(name)
```

ILXTransaction.setHeader

The ILXTransaction.setHeader method adds a header, or replaces an existing header of the same name; available in either a response or a request object.

```
ILXTransaction.setHeader(name, value)
```

ILXTransaction.replaceBody

The ILXTransaction.replaceBody method discards the body data; available in either a response or a request object. This method can be called after receipt of the `requestStart` or `responseStart` events and prior

to calling ILXTransaction.complete. The plug-in may replace the body by using the ILXFlow.client.write or ILXFlow.server.write methods.

```
ILXTransaction.replaceBody()
```

The following tables list the properties for ILXTransaction request and response objects.

Property	Description
request.params.uri	Specifies the request URI.
request.params.method	Specifies the request method.
request.params.version	Specifies the protocol version.
request.params.path	Specifies the path portion of a URI.
request.params.query	Specifies the query portion of a URI.
request.params.headers	Specifies the set of headers in a request.

Property	Description
response.params.status	Specifies the response status value.
response.params.version	Specifies the protocol version.
response.params.headers	Specifies the set of headers in a response.
response.params.closeClient	Indicates whether the ILX framework will automatically close the connection to the client when `response.complete` is called. Defaults to true. To change the behavior, set this property to false before calling `response.complete`.
response.params.closeServer	Indicates whether the ILX framework will automatically close the connection to the server when `response.complete` is called. Defaults to true. To change the behavior, set this property to false before calling `response.complete`.

iRulesLX streaming data native server code example

This code sample implements an HTTP server using native Node.js modules.

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

function httpRequest(req, res)
{
    res.end("got it: " + req.method + " " + req.url + "\n", "ascii");
}

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

iRulesLX streaming data pass-through HTTP server code example

This code sample implements a pass-through HTTP server. A virtual server must have an HTTP profile to handle the request start and response complete events that are common to HTTP requests.

```
var assert = require('assert');
var f5 = require('f5-nodejs');
var plugin = new f5.ILXPlugin();

function log(msg)
{
    if (plugin.globalTraceLevel() >= 5) {
        console.log(msg);
    }
}

plugin.on("connect", function(flow)
{
    flow.client.allow();
    flow.client.on("requestStart", function(request) {
        log("requestStart event");
        log("method: " + request.params.method);
        log("uri: " + request.params.uri);
        log("query: " + request.params.query);
        log("path: " + request.params.path);
        log("version: " + request.params.version);
        for (var hdr in request.params.headers) {
            log(hdr + ": " + request.params.headers[hdr]);
        }
    });
    flow.client.on("readable", function() {
        log("client readable event");
        var buf;
        while(true){
            buf = flow.client.read();
            if (buf !== null) {
                log("client body: " + buf.length + " bytes");
                log(buf.toString());
                flow.server.write(buf);
            }
            else {
                log("client EOF");
                break;
            }
        }
    });
    flow.client.on("requestComplete", function(request) {
        log("request complete: " + request.params.uri);
        log("request truncated: " + request.params.truncated);
        request.complete();
    });
    flow.server.on("connect", function() {
        log("server connect event");
    });
    flow.server.on("responseStart", function(response) {
        log("responseStart event");
        log("status: " + response.params.status);
        log("version: " + response.params.version);
        for (var hdr in response.params.headers) {
            log(hdr + ": " + response.params.headers[hdr]);
        }
```

```
    });
    flow.server.on("readable", function() {
        log("server readable event");
        var buf;
        while (true) {
            buf = flow.server.read();
            if (buf !== null) {
                log("server body: " + buf.length + " bytes");
                log(buf.toString());
                flow.client.write(buf);
            }
            else {
                log("server EOF");
                break;
            }
        }
    });
    flow.server.on("responseComplete", function(response) {
        log("response done event: " + response.params.status);
        log("response truncated: " + response.params.truncated);
        response.complete();
    });
});

var options = new f5.ILXPluginOptions();
options.handleClientOpen = true;
plugin.start(options);
```

iRulesLX streaming data read-write socket code example

This code sample uses the F5 Node.js module to read data from a server into a buffer, then writes the data to a client. Data received from a client is written into a buffer and then written to a server.

```
var f5 = require("f5-node.js");
var plugin = new f5.ILXPlugin();

plugin.on("connect", function(flow)
{
    flow.client.on("data", function(buffer) {
        flow.server.write(buffer);
    });
    flow.client.on("error", function(err) {
        console.log("client socket error: ", err);
    });
    flow.server.on("readable", function() {
        var buffer;
        while (true) {
            buffer = flow.server.read();
            if (buffer === null) {
                break;
            }
            flow.client.write(buffer);
        }
    });
    flow.server.on("error", function(err) {
        console.log("server socket error: " + err);
    });
    flow.on("error", function(err) {
        console.log("flow error: " + err);
    });
});
var options = new f5.ILXPluginOptions();
plugin.start(options);
```

iRulesLX streaming data server code example

This code sample creates a server that responds to clients as an HTTP server. The structure of the code makes it suitable to types of servers other than just HTTP.

```
var f5 = require('f5-nodejs');
var plugin = new f5.ILXPlugin();

plugin.on("connect", function(flow)
{
    flow.client.on("data", function(buffer) {
        flow.client.end(
            "HTTP/1.0 200 OK\r\n" +
            "Server: BigIP\r\n" +
            "Connection: Keep-Alive\r\n" +
            "Content-Length: " + 4 + "\r\n\r\n" +
            "abc\n");
    });
});
var options = new f5.ILXPluginOptions();
options.disableServer = true;
plugin.start(options);
```

Legal Notices

Legal notices

Publication Date

This document was published on Feb 13, 2017.

Publication Number

MAN-0591-01

Copyright

Trademarks

For a current list of F5 trademarks and service marks, see *http://www.f5.com/about/guidelines-policies/trademarks/*.

All other product and company names herein may be trademarks of their respective owners.

Patents

This product may be protected by one or more patents indicated at: *https://f5.com/about-us/policies/patents*

Export Regulation Notice

This product may include cryptographic software. Under the Export Administration Act, the United States government may consider it a criminal offense to export this product from the United States.

RF Interference Warning

This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures.

FCC Compliance

This equipment has been tested and found to comply with the limits for a Class A digital device pursuant to Part 15 of FCC rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This unit generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

Any modifications to this device, unless expressly approved by the manufacturer, can void the user's authority to operate this equipment under part 15 of the FCC rules.

Canadian Regulatory Compliance

This Class A digital apparatus complies with Canadian ICES-003.

Standards Compliance

This product conforms to the IEC, European Union, ANSI/UL and Canadian CSA standards applicable to Information Technology products at the time of manufacture.

Index