
Scripting Developer's Guide

- [Scripting Developer's Guide](#)
 - [About This Scripting Developer's Guide](#)
 - [Development Resources](#)
 - [Creating Scripts](#)
 - [Where Scripts Fit into the Data Path](#)
 - [Understanding Script Execution](#)
 - [When Script Changes Take Effect](#)
 - [Best Practices](#)
 - [Script Errors](#)
 - [Testing, Debugging, and Troubleshooting Scripts](#)
 - [Diagnosing Strange Script Behavior](#)
 - [A Variable Is Unset That I Know I Set](#)
 - [External Services Only Work Intermittently from a Script](#)
 - [My Compression Streams Never Complete](#)
 - [My Request Handler Is Only Called Sometimes](#)
 - [My Script Can Only Forward New Requests Sometimes](#)
 - [User Sessions Are Spuriously Deleted](#)
 - [Managing Scripts Using the REST API](#)

Scripting Developer's Guide

Overview

This guide describes the how to use the LineRate Scripting, best practices, debugging, and troubleshooting, as well as examples. This guide and the [Scripting API Reference Guide](#) describe the latest version of the LineRate Scripting, with current updates. Your version may be slightly different.

To access the LineRate Scripting reference information for the version you are running, access this REST API node on your system:

`https://<ip_address>:8443/doc/scripting/api`

Contents

The guide is broken into the following sections:

- [About This Scripting Developer's Guide](#)
- [Development Resources](#)
- [Creating Scripts](#)
- [Where Scripts Fit into the Data Path](#)
- [Understanding Script Execution](#)
- [When Script Changes Take Effect](#)
- [Best Practices](#)
- [Testing, Debugging, and Troubleshooting Scripts](#)
- [Diagnosing Strange Script Behavior](#)
- [Managing Scripts Using the REST API](#)

About This Scripting Developer's Guide

1. [Overview](#)
2. [Audience](#)
3. [Conventions](#)
4. [Searching the Guide](#)
 - 4.1. [Relevance Level](#)
 - 4.2. [Limiting a Search to Specific Tree](#)
 - 4.3. [Term Modifiers](#)
 - 4.3.1. [Wildcard Searches](#)
 - 4.3.2. [Fuzzy Searches](#)
 - 4.3.3. [Proximity Searches](#)
 - 4.3.4. [Boosting a Term](#)
 - 4.4. [Boolean Operators](#)
 - 4.4.1. [OR](#)
 - 4.4.2. [AND](#)
 - 4.4.3. [±](#)
 - 4.4.4. [NOT](#)
 - 4.5. [Grouping](#)
 - 4.6. [Escaping Special Characters](#)
5. [Example IP Addresses](#)
6. [Legal Notices](#)
 - 6.1. [Copyright](#)
 - 6.2. [Trademarks](#)

Overview






This About page contains general information about this guide, including the audience, typographic conventions, and how to search the content.

Audience

This guide is intended for experienced network administrators and network architects who understand your organization's existing TCP/IP network and who need to write scripts to customize how LineRate works in specific situations.

Conventions

This guide uses the following symbols and typographic conventions.

Convention	Definition
Monospaced bold	Text in a monospaced bold font represents commands or other text that you type exactly as you see it.
<angle brackets="brackets">	Text in a monospaced bold font inside angle brackets represents a placeholder that describes what you must type.
[square brackets]	Text in a monospaced bold font inside square brackets represents an optional command or option.
Monospaced	Text in a monospaced font represents output or results the system displays.
Bold	Text in bold shows keys to press and items to select or click, such as menu items or buttons.
	Shows the beginning of a procedure.
 Caution	Cautions contain critical information about configuring your system or data.
 Note	Notes contain important information that may affect how you install or configure your system.
 Tip	Tips contain best practices or useful information to help you when configuring your system.
	Shows that the content is for advanced users.

Searching the Guide

The search box at the top-right of each page lets you enter a term or phrase to search for. By default, the system searches all pages in the LineRate content. Searches are not case sensitive. By default, searches find plurals and other matches from word stems, such as tests, testing, tested, and tester if you search for test.

You can search for a single term such as:

```
interface
```

Or

```
certificate
```

You can also search for an exact phrase surrounded by double quotes such as:

```
"real server"
```

Or

```
"IP address"
```

Relevance Level

By default, the system sorts the search results by relevance. The relevance is determined by a weighting algorithm that takes into consideration the page title, content, tags, and attachments. The relevance is also affected by the page rating (thumb up or down) and by how often other users select a page to view from similar searches.

Searches can return a large number of results. You can narrow your searches a number of ways by:

- Limiting your search to a specific tree
- Using term modifiers
- Using Boolean operators

Limiting a Search to Specific Tree

If you only want to search one area or tree of a guide, you can limit your search to that tree. For example, if you only want to search the Configure Command tree of the 2.2 Release of the CLI Reference Guide for the term "interface," you can enter your search like this:

```
+(path:097Release_2.4/200CLI_Reference_Guide/Configure_Commands/*) AND interface
```

You can further narrow the search using the term modifiers and Boolean operators (described below):

```
+(path:097Release_2.4/200CLI_Reference_Guide/Configure_Commands/*) AND interface  
AND CARP
```

```
+(path:097Release_2.4/*) AND load AND balancer
```



Note: For a tree-specific search, words in quotes are not treated as a specific phrase. The search does an OR search for any words in quotes, so you may not want to use quotes and use AND instead, as shown in the example above.

A few steps to help with this type of search:

1. Navigate to the tree you want to search.
2. In your browser's address bar, copy the address of the page.
 - You only need the part after the "https://docs.lineratesystems.com/".
3. Using the syntax example above, type in your search and paste in the path of the page you want to search.

Term Modifiers

The search supports modifying query terms to provide a wide range of searching options.

Wildcard Searches

The guides support single- and multiple-character wildcard searches with single terms (not within phrase queries).

To perform a single-character wildcard search, use the ? symbol.

To perform a multiple-character wildcard search, use the * symbol.

The single-character wildcard search looks for terms that match that with the single character replaced. For example, to search for "text" or "test" you can use the search:

`te?t`

The multiple-character wildcard search looks for 0 or more characters. For example, to search for test, tests or tester, you can use the search:

`test*`

You can also use the wildcard searches in the middle of a term.

`te*t`



Note: You cannot use a * or ? symbol as the first character of a search.

Fuzzy Searches

The guide supports fuzzy searches based on the Levenshtein Distance or Edit Distance algorithm. To do a fuzzy, search use the tilde ~ symbol at the end of a single word. Fuzzy searches work for multiple characters. For example, to search for a term similar in spelling to "roam" use the fuzzy search:

```
roam~
```

This search will find terms like foam and roams.

You can add an optional parameter to specify the required similarity. The value is between 0 and 1. With a value closer to 1, only terms with a higher similarity will be matched. For example:

```
roam~0.6
```

The default is 0.5.

Proximity Searches

The guide supports finding words that are within a specific distance from each other. To do a proximity search, use the tilde ~ symbol at the end of a phrase. For example, to search for a "feature" and "standard" within 10 words of each other in a document use the search:

```
"feature standard"~10
```

Boosting a Term

The guide provides the relevance level of matching documents based on the terms found. To boost a term, use the caret ^ symbol with a boost factor (a number) at the end of the term you are searching. The higher the boost factor, the more relevant the term will be.

Boosting allows you to control the relevance of a document by boosting its term. For example, if you are searching for:

```
mindtouch search
```

and you want the term "mindtouch" to be more relevant boost it using the ^ symbol along with the boost factor next to the term. You would type:

```
mindtouch^4 search
```

This will make documents with the term mindtouch appear more relevant. You can also boost phrases as in the example:

```
"mindtouch search"^4 "Apache"
```

By default, the boost factor is 1. Although the boost factor must be positive, it can be less than 1 (e.g. 0.2)

Boolean Operators

Boolean operators allow terms to be combined through logic operators. MindTouch supports AND, +, OR, NOT, and - as Boolean operators.



Note: Boolean operators must be ALL CAPS.

OR

The OR operator is the default conjunction operator. This means that if there is no Boolean operator between two terms, the OR operator is used. The OR operator links two terms and finds a matching document if either of the terms exist in a document. This is equivalent to a union using sets. The symbol || can be used in place of the word OR.

To search for documents that contain either "mindtouch search" or just "mindtouch" use the query:

```
"mindtouch search" mindtouch
```

or

```
"mindtouch search" OR mindtouch
```

AND

The AND operator matches documents where both terms exist anywhere in the text of a single document. This is equivalent to an intersection using sets. You can use the symbol && in place of the word AND.

To search for documents that contain "mindtouch search" and "Advanced" use the query:

```
"mindtouch search" AND "Advanced"
```

+

The + (required operator) requires that the term after the + symbol exist somewhere in a document.

To search for documents that must contain "search" and may contain "advanced," use the query:

```
+search advanced
```

NOT

The NOT operator excludes documents that contain the term after NOT. This is equivalent to a difference using sets. You can use the symbol ! in place of the word NOT.

To search for documents that contain "mindtouch search" but not "Advanced" use the query:

```
"mindtouch search" NOT "Advanced"
```



Note: The NOT operator cannot be used with just one term. For example, the following search will return no results:

```
NOT "mindtouch search"
```

Grouping

The guide supports using parentheses to group clauses to form sub queries. This can be very useful if you want to control the Boolean logic for a query.

To search for either "mindtouch" or "search" and "advanced" use the query:

```
(mindtouch OR search) AND advanced
```

This eliminates any confusion and makes sure you that website must exist and either term mindtouch or search may exist.

Escaping Special Characters

The Guide supports escaping special characters that are part of the query syntax. The current list of special characters is:

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : \
```

To escape these character use the \ before the character. For example, to search for (1+1):2 use the query:

```
\(1\+1\)\:2
```

Example IP Addresses

Throughout this guide, we use example IP addresses for both internal (private) and external (public) uses.

For private addresses, we use the IP addresses designated in [RFC 1918](#):

- 10.0.0.0 - 10.255.255.255 (10/8 prefix)
- 172.16.0.0 - 172.31.255.255 (172.16/12 prefix)
- 192.168.0.0 - 192.168.255.255 (192.168/16 prefix)

For public addresses, we use the IP addresses designated for documentation in [RFC 5737](#):

- 192.0.2.0/24 (TEST-NET-1)
- 198.51.100.0/24 (TEST-NET-2)
- 203.0.113.0/24 (TEST-NET-3)

Legal Notices

Copyright

Copyright © 2014, F5 Networks, Inc. All rights reserved.

F5 Networks, Inc. (F5) believes the information it furnishes to be accurate and reliable. However, F5 assumes no responsibility for the use of this information, nor any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent, copyright, or other intellectual property right of F5 except as specifically described by applicable user licenses. F5 reserves the right to change specifications at any time without notice.

Trademarks

AAM, Access Policy Manager, Advanced Client Authentication, Advanced Firewall Manager, Advanced Routing, AFM, APM, Application Acceleration Manager, Application Security Manager, ARX, AskF5, ASM, BIG-IP, BIG-IQ, Cloud Extender, CloudFucious, Cloud Manager, Clustered Multiprocessing, CMP, COHESION, Data Manager, DevCentral, DevCentral [DESIGN], DNS Express, DSC, DSI, Edge Client, Edge Gateway, Edge Portal, ELEVATE, EM, Enterprise Manager, ENGAGE, F5, F5 [DESIGN], F5 Certified [DESIGN], F5 Networks, F5 SalesXchange [DESIGN], F5 Synthesis, f5 Synthesis, F5 Synthesis [DESIGN], F5 TechXchange [DESIGN], Fast Application Proxy, Fast Cache, FirePass, Global Traffic Manager, GTM, GUARDIAN, iApps, IBR, Intelligent Browser Referencing, Intelligent Compression, IPv6 Gateway, iControl, iHealth, iQuery, iRules, iRules OnDemand, iSession, L7 Rate Shaping, LC, Link Controller, Local Traffic Manager, LTM, LineRate, LineRate Systems [DESIGN], LROS, LTM, Message Security Manager, MSM, OneConnect, Packet Velocity, PEM, Policy Enforcement Manager, Protocol Security Manager, PSM, Real Traffic Policy Builder, SalesXchange, ScaleN, Signalling Delivery Controller, SDC, SSL Acceleration, software designed applications services, SDAC (except in Japan), StrongBox, SuperVIP, SYN Check, TCP Express, TDR, TechXchange, TMOS, TotALL, Traffic Management Operating System, Traffix Systems, Traffix Systems (DESIGN), Transparent Data Reduction, UNITY, VAULT, vCMP, VE F5 [DESIGN], Versafe, Versafe [DESIGN], VIPRION, Virtual Clustered Multiprocessing, WebSafe, and ZoneRunner, are trademarks or service marks of F5 Networks, Inc., in the U.S. and other countries, and may not be used without F5's express written consent.

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

Development Resources

1. [Overview](#)
 2. [JavaScript API References](#)
 3. [JavaScript Programming Guides](#)
 4. [Node.js Guides and References](#)
 5. [LineRate Scripting](#)
 6. [Node Modules Path](#)
-

Overview

This page lists additional resources that will help you understand LineRate Scripting's underlying programming tools and the LineRate Scripting API.

JavaScript API References

LineRate Scripting uses the JavaScript programming language.

For more information about JavaScript API, refer to the following resources:

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/>
 - <http://www.ecma-international.org/ecma-262/5.1/>
-

JavaScript Programming Guides

For more information about general JavaScript programming and best practices, refer to the following resources:

- <https://developer.mozilla.org/en-US/docs/JavaScript/Guide>
 - [JavaScript: The Good Parts](#)
-

Node.js Guides and References

In addition to bare JavaScript, LineRate Scripting uses the Node.js (v0.8.3) libraries. If you are familiar with node.js, LineRate Scripting uses many of the node.js concepts and terminology. However, LineRate Scripting does have some differences from Node.js.

For more information about node.js, refer to the following resources:

- [Node.js](#)
- [The Node Beginner Book](#)

LineRate Scripting

LineRate Scripting includes additional libraries and modules similar to node.js, but that are unique to LineRate. Documentation for all of the libraries and modules LineRate Scripting uses for your version is available from the following REST API node:

`https://<ip_address>:8443/doc/scripting/api`

The latest version of the LineRate Scripting documentation is also available from the [Scripting API Reference Guide](#).



Tip: We strongly recommend that you gain a good understanding of events in node.js, before writing any scripts.

You can create scripts using both the CLI and the REST API.

Node Modules Path

You can extend the functionality of the LineRate Scripting by using JavaScript modules. Using these modules follows the standards defined in <http://nodejs.org/api/modules.html>. The LineRate Scripting will search for **node_modules** directories starting with a root path of **/home/linerate/data/scripting** .

Creating Scripts

1. [Overview](#)
 2. [Example Use Cases](#)
 3. [Understanding Script "Mechanics"](#)
 4. [What Happens After Creating a Script](#)
 5. [Creating Your First Script Using the CLI](#)
 6. [First Script Example for Copying](#)
 7. [Complete Show Run](#)
-

Overview

The LineRate Scripting lets you create custom scripts to manage, redirect, and control the requests and responses in your traffic flow. Most of the examples in this Scripting Developer's Guide use forward proxies. However, you can use scripts with both forward proxy and reverse proxy configurations.

Example Use Cases

Following are just a few examples of common script use cases:

- Manipulate HTTP request and response headers to do such things as:
 - Delete cookies for security reasons.
 - Delete response headers that identify servers in your network.
 - Add headers to trace the path through your network
 - Add headers for personalization.
 - Add the proxy to the header string to track which proxies a request has been routed through.
- Query external resources to do such things as:
 - Send data about the client to a server to personalize the response.
 - Authenticate the client.
- Steer requests and responses based on their content to do such things as:
 - Send a request to a local video cache.
 - Send a request to local content optimizer.
- Respond directly to a request to do such things as:
 - Construct a redirect to send requests for a specific server, perhaps during maintenance, to a page that tells the user the server status.
 - Apply policies to restrict access to certain sites or types of content.
- Transform the body of requests and responses to do such things as:
 - Replicate traffic (dark traffic) to perform A/B testing.

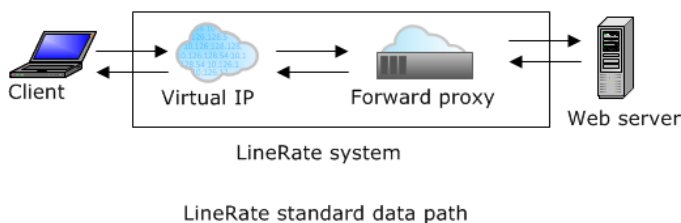
- Parse request content to send portions of the content to different databases.

Understanding Script "Mechanics"

You can think of scripts as intercepting a request or response, thereby disconnecting the data path, then sending the request or response on its way. It's like a detour, and you have to explicitly say where the script should send the request or response after running the script.

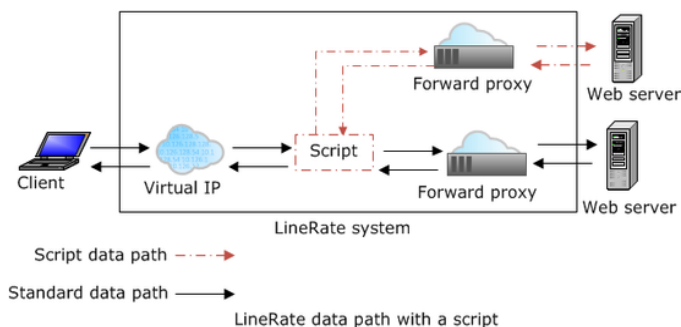
You can call scripts at specific points in the data path, called data path events. The data path events are specific types of events where a script can intercept a request or response in the data path.

The standard data path looks like the diagram below.



In the standard data path, the request comes in from the client, goes through the virtual IP, gets passed to a forward proxy, and finally to the web server. The response follows the same path back to the client.

An example of the data path with a script could look like the diagram below.



In this example, the script "intercepts" the request just before it reaches the forward proxy. The script sends the request to a different forward proxy from the standard data path, based on what the script code contains, then the request continues on to a web server. Notice that the response takes the same route back to the client.

This script data path shows an example of steering the request based on a URL. The script sees that the requested URL matches what it is coded to look for, so the script intercepts the request and passes it on to a different forward proxy and web server.

For more information about the data path events, see [Where Scripts Fit into the Data Path](#).

What Happens After Creating a Script

After you create a script and set its admin status to online:

- The script runs to completion by executing the JavaScript statements in the script. (You can think of this as an initialization process.)
- A JavaScript statement may register a listener for an event, unregister a listener for an event that was previously registered, emit a custom event, or do something else.
 - An event listener is a JavaScript function containing statements that will be executed whenever an event occurs. An event may have 0, 1, or many listeners.
- Whenever an event occurs, the event listeners for that event are executed (run to completion).
- The set of listeners and events are dynamically changing throughout the lifetime of the script.
 - Some events, like configuration events, may occur repeatedly and rarely, and usually have a small set of listeners that are permanently associated.
 - Some events, like the 'response' event on an individual HTTP request, are only fired once. Therefore, their event listeners are only invoked once and are usually short-lived.

In general, scripts can register for any of the events in the supported modules of Node.js. Most LineRate scripts register for events that interact with the configuration of the LineRate system. These events have special behaviors related to when the object (such as a virtual server or forward proxy) is created. In general:

- If an object called for in the script already exists and is online, the 'exist' event listener for the object will execute immediately and the next request will be diverted into the event listener for the object. Note that even if you create the object, then immediately create the script, some requests could be flowing through the object before the script is available to intercept the requests.
- If the object does not exist yet, the 'exist' event listener will be invoked as soon as the object is created, set to admin-status online, and a request comes into object.

The description of each event in the [Scripting API Reference Guide](#) includes details of what occurs if objects acted upon in the script already exist or are created later.

Creating Your First Script Using the CLI

Using the CLI, you can type or copy and paste a script directly into the command line, or you can call a file that contains the script.

For longer scripts, more than about 20 lines, we recommend using a file for easier maintenance.

In this short example, the script creates a function that is called whenever a request is directed to the forward proxy called myForwardProxy. For just the script that you can copy and paste, see [First Script Example for Copying](#).



To create and load a script directly from the command line:

Command	Description
<code>configure</code>	Puts LineRate into configure mode.
<code>forward-proxy myForwardProxy attach virtual-ip vip1 admin-status online</code>	<p>Creates a forward proxy call myForwardProxy.</p> <p>Attaches the virtual IP called vip1 to myForwardProxy.</p> <p>Puts the forward proxy online.</p>
<code>virtual-ip vip1 ip range 0.0.0.0 255.255.255.255 8080 admin-status online</code>	<p>Creates the virtual IP called vip1.</p> <p>Sets the virtual IP range from 0.0.0.0 to 255.255.255.255, effectively including every IPv4 possible, on port 8080.</p> <p>Puts the virtual IP online.</p>
<code>script fproxy-add-proxyhost</code>	Puts LineRate into script mode and creates a script called fproxy-add-proxyhost.
<code>source inline "ENDSCRIPT"</code>	Sets the script source to be "inline" (not from a file) and sets the word ENDSCRIPT as the word that denotes the end of script code. The next line starts the actual script.
<code>"use strict";</code>	We recommend using strict mode for all scripts. See Use JavaScript Strict Mode .
<code>var fpm = require('lrs/ forwardProxyModule');</code>	Loads the LineRate Scripting library called forwardProxyModule and stores it in the object we called fpm.
<code>var os=require('os'); var proxyhost = os.hostname();</code>	<p>Loads the LineRate Scripting library called os and stores it in the object called os.</p> <p>Loads the LineRate Scripting library called os.hostname and stores it in the object called proxyhost.</p>
<code>var onRequest= function(servReq,servResp, cliReq) { servReq.addHeader('X-Proxy- Host', proxyhost); cliReq(); };</code>	<p>Declares a function named onRequest with arguments of request, a response, and cliReq.</p> <ul style="list-style-type: none">The servReq argument lets the function access the client's HTTP request.

Command	Description
	<ul style="list-style-type: none"> • The servResp argument lets the script access the response from the backend server. • The cliReq object lets the script pass the request back into the traffic flow, just as the request would have had the function not been called. <p>The servReq.addHeader (a LineRate Scripting method) inserts the header called X-Proxy-Host. The header is a custom HTTP header containing the hostname of the LineRate system.</p> <p>Then the script calls the object called cliReq() to continue sending the response.</p>
<pre>var onExist = function(fp) { fp.on('request', onRequest); console.log('Forward Proxy ' + fp.id + ' exists. '); };</pre>	<p>Declares a function called onExist with the argument fp (forward proxy).</p> <p>The fp.on function calls the onRequest function. Whenever a request comes in for the forward proxy called myForwardProxy, send the request to the function called onRequest. This creates the interception of the request.</p> <p>The console.log line logs the string "Forward Proxy" following by the name of the forward proxy called myForwardProxy, followed by the string "exists." Note that this console.log line will run on each load balancing process on the system each time the script starts.</p>
<pre>fpm.on('exist', 'myForwardProxy', onExist);</pre>	<p>Registers an event handler (also called a listener) that waits for the forward proxy called myForwardProxy to exist. When it does exist, it calls the function called onExist.</p> <p>The first time the script runs, this is the only part that actually does anything.</p>
ENDSCRIPT	<p>This token tells the system that this is the end of inline script source. This also loads the script and compiles it. The compiler will report any errors.</p>
admin-status online	<p>Sets the script's admin status to online, so the script is ready to run whenever a request comes into the forward proxy called myForwardProxy.</p>

First Script Example for Copying

Copy and paste the script below to try it.

```
"use strict";
var fpm = require('lrs/forwardProxyModule');
var os = require('os');
var proxyhost = os.hostname();
var onRequest= function(servReq, servResp, cliReq) {
  servReq.addHeader('X-Proxy-Host', proxyhost);
  cliReq();
};
var onExist = function(fp) {
  fp.on('request', onRequest);
  console.log('Forward Proxy ' + fp.id + ' exists.');
```

```
};
fpm.on('exist', 'myForwardProxy', onExist);
```

Complete Show Run

Below is the complete **show run** output for this example, including the entire configuration needed.

```
!
hostname example-host
!
username admin secret encrypted
"$2a$04$aYtlW5E6BltzZ0pczMB0veX6qeTymcU.sOIItLP1PbiLr1H24YF6u2" uid 2000
!
interface em0
ip address 10.126.16.127 255.255.0.0
!
ssl profile self-signed
attach primary-certificate self-signed
attach private-key self-signed
!
forward-proxy myForwardProxy
attach virtual-ip vipl
admin-status online
!
virtual-ip vipl
ip range 0.0.0.0 255.255.255.255 8080
admin-status online
!
ssh
allow from any
allow to any 22
!
rest-server
allow from any
allow to any 8443
attach ssl profile self-signed
!
script fproxy-add-proxy-host
```

```
source inline "ENDSOURCE"
var fpm = require('lrs/forwardProxyModule');
var os = require('os');
var proxyhost = os.hostname();
var onRequest= function(servReq, servResp, cliReq) {
console.log("PID: " + process.pid + " request: " + request.url); // log the request to
servReq.addHeader('X-Proxy-Host', proxyhost);
cliReq();
}
var onExist = function(fp) {
fp.on('request', onRequest);
console.log('Forward Proxy ' + fp.id + ' exists.');
```

```

}
fpm.on('exist', 'myForwardProxy', onExist);
ENDSOURCE
```

admin-status online

!

license-manager

ip 1 10.126.64.38

!

Where Scripts Fit into the Data Path

1. [Overview](#)
2. [Script Events](#)
3. [Request and Response Objects](#)
4. [Script Data Path Events](#)
5. [onServerRequest](#)
 - 5.1. [Continue on Original Data Path](#)
 - 5.2. [Steer Request to Another Configured Object in the System](#)
 - 5.3. [Generate the Response in the Script](#)
6. [onClientResponse](#)
 - 6.1. [Continue on Original Path](#)
 - 6.2. [Remove or Change Response Headers](#)

Overview

This page describes script events, in general, and LineRate data path events.

Script Events

Scripts register for events. Node.js and JavaScript define many types of common events (such as timers, completion of background jobs, IO, and unhandled error notifications) that scripts can register for.

Scripts can also define their own events that they register and listen for. The LineRate modules define custom events that interact with the data path.

The remainder of this page describes the LineRate data path events.

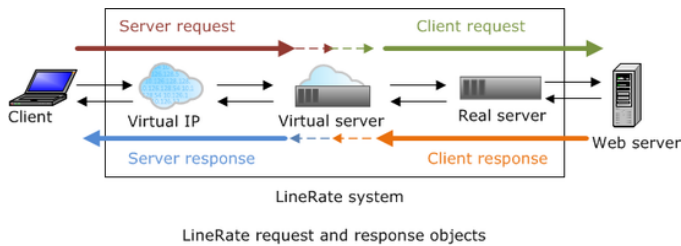
Request and Response Objects

The LineRate Scripting follows the Node.js structure for requests and response. The LineRate Scripting uses four types of objects that are passed to the data path event listeners:

- Server request
- Client request
- Client response
- Server response

The diagram below depicts these four objects. The server vs. client depends on the relationship of the request or response to the LineRate server. For example, when a user loads a web page, the request coming into LineRate is a server request, with LineRate acting as the server at this point. As the request passes through the virtual server in LineRate, it becomes a client request, because LineRate is the client to the web server.

Responses work the same way, only going in the opposite direction.

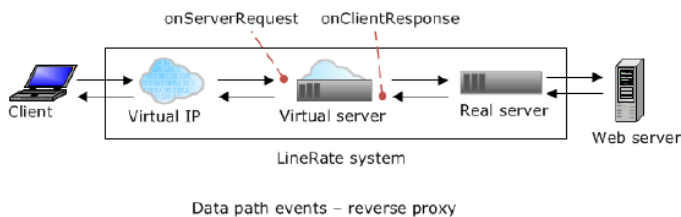
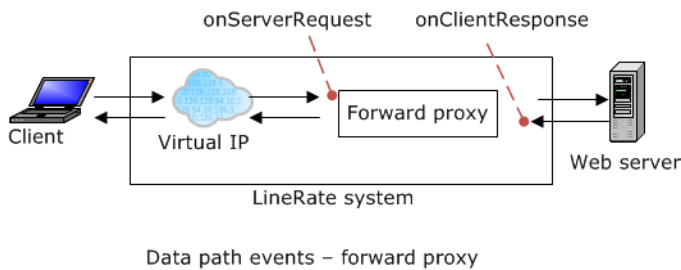


Script Data Path Events

You can think of data path events as "attach points" in the data path. A script diverts the request or response at a data path event, performs its function, then sends the request or response on its way. The data path events occur at specific points in the data path.

The diagrams below shows the available data path events that you can use with the LineRate Scripting:

- onServerRequest
- onClientResponse



onServerRequest

The onServerRequest data path event intercepts the request on its way from the client before it reaches the virtual server and takes one or more of the following types of actions:

- Continue on original path
- Steer request to another configured object in the system
- Generate a response

You can also create a script using a combination of all three of these actions.

The API for handling a request is:

```
var onServerRequest=function(servReq, servResp, cliReq) {
```

- The **servReq** argument lets the script manipulate the request from the client.
- The **servResp** argument lets the script create a response.
- The **cliReq** argument lets the script manipulate the request that is sent to the web server.

Continue on Original Data Path

After intercepting the request, your script may look for something (specific URL or data type, such as video). If the request does not contain anything the script needs to act upon, your script can tell the request to continue on its original data path. To select this action, call **cliReq()**.

```
var forwardUntouched=function(servReq, servResp, cliReq) {  
    cliReq();  
};
```

Steer Request to Another Configured Object in the System

After intercepting the request, your script can look for something (specific URL or data type, such as an image). If the request does contain what the script is looking for, for example an image, your script can steer the request to an image processing server. For this use case, users are accessing images on your server. For users who are not registered with your site (this part of the script is not shown), you want to add a water mark to the image before displaying it.

To direct the request to an image processing virtual server to add the watermark, call **vs.moveRequest** and give it the request and response objects.

Script	Description
--------	-------------

<code>var vs = vsm.find('vs-images');</code>	Loads the LineRate Scripting library called <code>vsm.find</code> , which looks for a virtual server called <code>vs-images</code> , and stores it in the object called <code>vs</code> .
<code>var pattern = /\.(jpe?g png gif)/;</code>	Creates a pattern search for requested files that end in <code>.jpg</code> , <code>.jpeg</code> , <code>.png</code> , or <code>.gif</code> .
<pre>function forwardAlong(servReq, servResp, cliReq) { if (pattern.test(servReq.url)) { console.log("Watermarking image at " + servReq.url); vs.moveRequest(servReq, servResp, cliReq); } else {</pre>	<p>Declares the function called <code>forwardAlong</code> with the arguments for the request, response, and <code>cliReq</code> objects.</p> <p>If the pattern search matches:</p> <ul style="list-style-type: none"> • Logs the request to the <code>console.log</code> file. • Passes the request to the <code>vs-images</code> virtual server.
<pre>cliReq(); };</pre>	If the pattern search does not match, calls the function called <code>cliReq</code> to continue sending the response.

Below is the complete snippet that you can copy and paste:

```
var vs = vsm.find('vs-images');
var pattern = /\.(jpe?g|png|gif)/;
function forwardAlong(servReq, servResp, cliReq) {
  if (pattern.test(servReq.url)) {
    console.log("Watermarking image at " + servReq.url);
    vs.moveRequest(servReq, servResp, cliReq);
  } else {
    cliReq();
  }
};
```

Generate the Response in the Script

After intercepting the request, your script can look for some attribute of the request (such as a specific URL or the time of day). If the request does meet the script's criteria, for example a site blocked by parental controls, your script can generate the response saying that the URL is blocked. To write the response, call `servResp.write` and give it the request and response objects.

Script	Description
--------	-------------

<pre>function onRequest(servReq, servResp, cliReq) { if (checkBlockedHours(servReq) === true) { writeBlockedMessage('Blocked due to time of day restrictions', servReq, servResp, cliReq); } else { cliReq(); }; };</pre>	<p>Declares the function called onRequest with the arguments for the request, response, and cliReq objects.</p> <p>The if statement checks whether the request is within the configured checkBlockedHours (setting some time of day restrictions using parental controls). If checkBlockedHours returns true, the request is passed on the writeBlockedMessage function. If it is not true, the script calls the function called cliReq to send the request on its way.</p> <p>The writeBlockedMessage function has the arguments</p>
<pre>function writeBlockedMessage(title, servReq, servResp, cliReq) { servResp.setHeader('Content-Type', 'text/ html'); servResp.writeHead(450, 'Parental Controls', {}); servResp.write('<html><head><title>' + title + '</title></head><body> ' + </body></html> ' ' + title + ' Parental Controls has blocked access to this site. ' + ' If you feel that you have arrived at this page by mistake, please contact your ' + 'Administrator ' + '</pre>	<p>Declares the function that we called writeBlockedMessage with the arguments of title (which will be used as both the HTML title tag and in displayed in the body of the page), servReq, servResp, and cliReq.</p> <p>The writeBlockedMessage calls the servResp.setHeader, servResp.writeHead, and servResp.write methods to create an HTTP response containing a header with an HTTP 450 error code and the HTML page explaining the error.</p> <p>The last line defines the end of the HTML page that will become the server response.</p>
<pre>cliReq(); };</pre>	<p>Calls the object called cliReq to continue sending the response.</p>

Below is the complete snippet that you can copy and paste:

```
function onRequest(servReq, servResp, cliReq) {
    if (checkBlockedHours(servReq) === true) {
        writeBlockedMessage('Blocked due to time of day restrictions', servReq, servResp, cliReq);
    } else {
        cliReq();
    };
};

function writeBlockedMessage(title, servReq, servResp, cliReq) {
    servResp.setHeader('Content-Type', 'text/html');
    servResp.writeHead(450, 'Parental Controls', {});
    servResp.write('<html><head><title>' + title + '</title></head><body><p>' +
        '<h1><font color="red">' + title + '</font></h1> <p><br> Parental Controls has blocked access
to this site. <br>' +
        '<br> If you feel that you have arrived at this page by mistake, please contact your <a
href="mailto:administrator@example.com">' +
        'Administrator </a>' +
        '</p></body></html>');
    servResp.end();
    cliReq();
};
```

onClientResponse

The `onClientResponse` data path event intercepts the response on its way from the back-end server before it reaches the virtual server and takes one or more of the following types of actions:

- Continue on original path
- Remove or change response headers
- Create a new request and response

You can also create a script using a combination of all three of these actions.

The API for handling a response is:

```
servReq.on('response', function onResp(cliResp) {
    // Make the server response have the same headers as the client response
    cliResp.bindHeaders(servResp);

    // Do something interesting to the headers or response

    // Make the body of the server response the same as the body of
    // the client response.
    cliResp.pipe(servResp);
});
```

- The **`servReq`** object lets the script manipulate the request from the client.
- The **`cliResp`** argument lets the script manipulate the client response.
- The **`servResp`** object lets the script create a response.

Continue on Original Path

After intercepting the response, your script may look for something, such as a specific header. If the response does not contain what the script is looking for, your script can tell the response to continue on its original data path. To select this action, call `bindHeaders()` and then `pipe()`:

```
var forwardUntouched = function(cliResp, servResp) {  
  cliResp.bindHeaders(servResp);  
  cliResp.pipe(servResp);  
};
```

Remove or Change Response Headers

After intercepting the response, your script can look for one or more HTTP headers and either remove or change the header. For example, by default Server headers in the response tell the client the name and version of the server process (for example, "Server: Apache/2.4.1 (Unix)"). This can be a security risk, because it tells potential attackers which exploits might work on your server. Remove or change response headers by registering a listener for the `response` event, using `servReq.on('response', ...)`.

Script	Description
<pre>function onRequest(servReq, servResp, cliReq){ servReq.on('response', function onResp(cliResp) { cliResp.bindHeaders(servResp); servResp.removeHeader('Server'); cliResp.pipe(servResp); }); };</pre>	<p>Declares the function called <code>onRequest</code> with the arguments for the request, response, and <code>cliReq</code> objects.</p> <p>Calls the <code>servReq.on</code> method with the arguments 'response', which is the event which calls the callback, the function <code>onResp</code>, the function to run when this callback is triggered.</p> <p>Calls the <code>cliResp.bindHeaders</code> method that efficiently binds the client response headers to the server response headers, so that any changes to the server response headers are also made to the client response headers.</p> <p>Calls the <code>servResp.removeHeader</code> method, which removes any instances of the header called 'Server'.</p>

	<p>Calls the cliResp.pipe method which pipes the changed server response header into the client response.</p>
<pre>}; cliReq();</pre>	<p>Calls the function called cliReq to continue after sending the response.</p>

Below is the complete snippet you can copy and paste:

```
var fpm = require('lrs/forwardProxyModule');

function onRequest(servReq, servResp, cliReq){
  servReq.on('response', function onResp(cliResp) {
    cliResp.bindHeaders(servResp);
    servResp.removeHeader('Server');
    cliResp.pipe(servResp);
  });
  cliReq();
};

var onExist = function(fp) {
  fp.on('request', onRequest);
  console.log('Forward Proxy ' + fp.id + ' exists.');
};
fpm.on('exist', 'myForwardProxy', onExist);
```

Understanding Script Execution

1. [Overview](#)
2. [Scripts Execute on Multiple Cores](#)
3. [Logging Implications](#)
4. [Logging Example](#)

Overview

LineRate runs data path processes on multiple processor cores for best performance. This enables multiple requests and scripts to be processed simultaneously.

Scripts Execute on Multiple Cores

Functions that are in the global scope are executed by all the processes.

Functions that are attached to one of the attach points are executed only one process:

- `onServerRequest`—When you use this data path attach point, the script executes only on the core that handles the request, therefore, this part of the script executes only once.
- `onClientResponse`—When you use this data path attach point, the script executes only on the core that handles the response, therefore, this part of the script executes only once.



Tip: When you create your scripts, consider whether it is appropriate to execute each function multiple times. For example, if you are processing payments or counting votes, you do not want the payment processing or vote counting portion of the script to execute multiple times. This execution on each process also affects performance.

Logging Implications

When you include a `console.log` line in your script, the script executes that line on each data path process. Therefore, `/var/log/controller.messages` includes output from each data path process. For information about `console.log`, see [Logging with Console.log](#).

For example, if you have five data path processes, the script runs on each of five CPU cores. This means that for each console.log line of code, you will see the logging five times in the console.log file. Logging while testing can show what the script is doing and whether it is executing as expected. However, it can also quickly create huge log files. This will impact performance, and could fill up a partition on the LineRate's disk. Logging to disk should only be used in production systems for very rare conditions, or while actively monitored.

Logging Example

The example below contains three console.log lines. The first and third execute on each core, but note that these portions of the script only execute when the script is first started. The scripts start you set the script to admin-status online, restart the system, first create the virtual IP, or create or enable the forward proxy.

Here's what this example script is doing. Below is what you would see in the console.log file.

Script	Description
<pre>conconsole.log('script add-proxy-host starting on process ' + process.pid);</pre>	<p>Calls console.log to add content to the /var/log/controller.messages file. This adds the string "script add-proxy-host starting on" and the ID of the core that runs the script. Note that this console.log line will run on each load balancing core on the system each time the script starts.</p>
<pre>var assert = require('assert'); var os = require('os'); var fpm = require('lrs/ forwardProxyModule'); var proxyhost = os.hostname();</pre>	<p>Loads the LineRate Scripting libraries called assert, os, lrs/forwardProxyModule, and os.host, and stores them in the objects listed after each var keyword.</p>
<pre>var onRequest= function(request, response, cliReq) { console.log('PID: ' + process.pid + ' request: ' + request.url); request.addHeader('X-Proxy-Host', proxyhost);</pre>	<p>Loads the library called onRequest that gets a request, a response, and an object called cliReq.</p> <p>Calls console.log to log information for each request the script processes. The log file will contain the string "PID:" followed by the process ID of the core that handled the request and the</p>

<pre>cliReq(); }</pre>	<p>string "request:" followed by the URL of the requested page. Note that this line logs each request that comes through myForwardProxy.</p> <p>The request.addHeader (a LineRate Scripting method) adds the header we called X-Proxy-Host. The header contains the name of the host running LineRate. Note that duplicate headers are possible with this method; to replace an existing header, use the request.setHeader method.</p> <p>Then the script calls the object called cliReq.</p>
<pre>var onExist = function(fp) { assert.equal(fp.id, 'myForwardProxy'); fp.on('request', onRequest); console.log('Forward Proxy ' + fp.id + ' exists.');</pre>	<p>Loads the library called onExist which sets up an event listener, and looks for requests that come through the forward proxy called myForwardProxy, then sends the request to the onRequest call.</p> <p>The console.log line logs the string "Forward Proxy" followed by the name of the forward proxy called myForwardProxy, followed by the string "exists." Note that this console.log line will run on each load balancing core on the system each time the script starts.</p>
<pre>fpm.on('exist', 'myForwardProxy', onExist);</pre>	<p>Registers an event handler (also called a listener) that says: whenever a request comes in for the forward proxy called myForwardProxy, send the request to the function called onExist. This creates the interception of the request.</p> <p>The first time the script runs, this is the only part that actually does anything.</p>

For this example, the system is running five data path processes. At startup, console output in `/var/log/controller.messages` contains the console.log output for the first and third console.log lines five times, once for each process . The messages are color-coded below to match the line in the script above that generates them.

```
Jan 29 12:10:16 example-host LROS: script add-proxy-host starting on process 1507
Jan 29 12:10:16 example-host LROS: Forward Proxy myForwardProxy exists.
Jan 29 12:10:16 example-host LROS: script add-proxy-host starting on process 1789
Jan 29 12:10:16 example-host LROS: script add-proxy-host starting on process 1790
Jan 29 12:10:16 example-host LROS: Forward Proxy myForwardProxy exists.
Jan 29 12:10:16 example-host LROS: Forward Proxy myForwardProxy exists.
Jan 29 12:10:16 example-host LROS: script add-proxy-host starting on process 1792
Jan 29 12:10:16 example-host LROS: script add-proxy-host starting on process 1791
Jan 29 12:10:16 example-host LROS: Forward Proxy myForwardProxy exists.
Jan 29 12:10:16 example-host LROS: Forward Proxy myForwardProxy exists.
```

As requests come through the system, the script generates one line per request, indicating that each request is handled by a single process.

```
Jan 29 12:59:29 example-host LROS: PID: 1789 request: /file.txt
Jan 29 12:59:30
```

When Script Changes Take Effect

1. [Overview](#)
 2. [When Script Changes Take Effect](#)
-

Overview

This page describes when both changes to a script take effect and newly created scripts start to run.

When Script Changes Take Effect

Exactly when changes to an existing script take effect depends on the script was edited or created, as described below. The following is also true for newly created scripts.

How/where script changes are made	Changes take effect ...
LineRate CLI (<code>script <name> edit</code>) or edit in LineRate Manager.	After you save and quit.
LineRate CLI (<code>script <name> edit</code>) or edit in LineRate Manager, but quit without saving.	Never. Script does not change and continues to run in its original form.
REST API (<code>/config/script/<name>/sourceFile</code> or <code>/config/script/<name>/sourceInline</code>).	After you complete the PUT or POST command.
Not recommended. Outside of LineRate. For example, edit a script source file in bash mode or overwrite the script file using scp.	After you set the script to admin-status offline , then back to admin-status online .

For information about how scripts interact with objects that already exist or are created, see [What Happens After Creating a Script](#).

Best Practices

1. [Overview](#)
 2. [Recommended Style Guidelines](#)
 - 2.1. [Use JavaScript Strict Mode](#)
 - 2.2. [Follow a JavaScript Style Guide](#)
 - 2.3. [Use 'exists' Event for Configuration Objects](#)
 3. [Performance Best Practices](#)
 - 3.1. [Do Not Require Modules after a Script Starts](#)
 - 3.2. [Always Create Object Properties in the Same Order](#)
 - 3.3. [Expensive Computations Should Be Distributed and Asynchronous](#)
-

Overview

The following sections describe some best practices for using the LineRate Scripting.

Recommended Style Guidelines

Below are recommended style guidelines (more about each in the sections that follow):

- Use JavaScript strict mode.
 - Follow a JavaScript style guide.
 - Use 'exists' event for configuration objects.
-

Use JavaScript Strict Mode

JavaScript strict mode makes several language "gotchas" into errors. To apply it to an entire script, make sure the first statement is the string "use script":

```
// Comments before "use strict" are OK
"use strict";

function callMe() {
  localVariable = 42; // Oops! This is actually global!
                    // But "use strict" caught it.
  return localVariable;
}
```

A good practice is to develop new scripts to be entirely strict-mode compatible. However, if you must mix legacy and new code, you can invoke strict mode only in new functions, too:

```
function oldCode() {
  //Old code that intentionally makes global variables from inner scopes
  brandNewGlobalVariable = 42;
}

function newCode() {
  "use strict";
  localVariable = 42; // Oops! This is actually global!
                      // But "use strict" caught it!
  return localVariable;
}
```

Strict mode is usually as fast or slightly faster in the V8 engine than non-strict mode. More details about strict mode are available from MDN's [Strict Mode](#) documentation.

Follow a JavaScript Style Guide

The [Google JavaScript Style Guide](#) is useful and is what we follow in our examples.

Use 'exists' Event for Configuration Objects

The Forward Proxy Module and Virtual Server Module expose two events for interacting with configured objects: the 'exist' event and the 'create' event. The events occur under different circumstances.

Event	'create' occurs	'exist' occurs
First configured while script is running	Yes	Yes
Deconfigured and reconfigured while script is running	Yes	Yes
Already configured when script starts	No	Yes

Scripts usually want to be invoked whenever a request arrives at a particular forward proxy or virtual server. The script usually does not care whether the forward proxy or virtual server is configured before or after the script. In these cases, 'exist' is the correct event to use. Mistakenly using 'create' can lead to a script bug that appears after a reboot. Consider the following counter-example:

```
var vsm = require('lrs/virtualServerModule');

// BAD: Listen for the 'exist' event instead
vsm.on('create', 'vs1', function (vs) {
  vs.on('request', function(req, resp, next) {
    resp.end('Here is a simple response');
    next();
  });
});
```

When first configured, suppose you take these steps, configuring the script before the virtual server:

```
lros(config)# script testScript
lros(config:script-testscript)# admin-status online
lros(config:script-testscript)# virtual-server vs1
```

The script will load and handle requests as intended. However, if you save the configuration ('write memory') and then reboot, the script will not handle requests, because the virtual server is configured before the script starts.

Performance Best Practices

Below are performance best practices (more about each in the sections that follow):

- Do not require modules after a script starts.
- Always create object properties in the same order.
- Expensive computations should be distributed and asynchronous.

Do Not Require Modules after a Script Starts

Using `require()` to load a new module blocks execution while the following steps occur:

1. If the module is present in the module cache because it has already been loaded, it is returned immediately.
2. The module system examines several paths on the filesystem to locate the module.
3. The module definition (if in a package.json file) is loaded from the disk and parsed. It points to a main module file.
4. The file is loaded from disk.
5. The file is compiled.
6. The file is executed. If the execution of the file necessitates loading other files (either from this module or from other modules), then steps 1-6 are recursively executed on all of those files and modules.
7. The result of the execution is returned to the script.

While these steps are being executed, the data path process is blocked and will not process any other traffic. Therefore, it is important to ensure that any `require()` statements execute when the script is first loaded. Here is an example:

```
// GOOD: require() statements are the first statements in the script, at outer scope.
var url = require('url');

function onNewUser(request, response, next) {
  console.log('A new user, request URL parts:', url.parse(request.url));
}
```

In contrast, this example is incorrect, because the process will block while the url module is loaded:

```
function onNewUser(request, response, next) {
  // BAD: process will block while require('url') executes
  console.log('A new user, request URL parts:', require('url').parse(request.url));
}
```

In the previous example, it is possible that the url module is in the cache and therefore easy to load. However, that may change due to future code maintenance. It is recommended to always `require()` all modules that might be needed at the top of the script.

Always Create Object Properties in the Same Order

JavaScript is a dynamic language, and the set of properties that an object has can change over the duration of the object's lifetime. However, if you are creating a new object that has a specific set of properties, it is more efficient if that specific set of properties is always applied in the same order.

Here is an example: Suppose you can create a new instance of the Student class via two functions, `newStudentFromDatabase()` and `newStudentFromWebpage()`. If the two methods look like the following, then the resulting objects `studentA` and `studentB` will have the same internal hidden class, and code that uses them will be faster:

```
function newStudentFromDatabase(db, studentId) {
  var student = {};
  student.id = studentId;
  student.name = db.findById(studentId).name;
  return student;
}
function newStudentFromWebpage(name) {
  var student = {};
  student.id = getNewStudentId();
  student.name = name;
  return student;
}
var studentA = newStudentFromDatabase(theDatabase, 14321);
var studentB = newStudentFromWebpage('John Smith');
// GOOD: studentA and studentB have the same internal class
```

However, if the methods, instead, look like the following, `studentA` and `studentB` will not have the same internal hidden class. Code that uses students will have to be optimized for the hidden class for `studentA` and then again optimized independently for the hidden class for `studentB`:

```
function newStudentFromDatabase(db, studentId) {
  var student = {};
  student.id = studentId;
  student.name = db.findById(studentId).name;
  return student;
}
function newStudentFromWebpage(name) {
  var student = {};

  // BAD: student.name and student.id are reordered
  student.name = name;
  student.id = getNewStudentId();
  return student;
}
var studentA = newStudentFromDatabase(theDatabase, 14321);
var studentB = newStudentFromWebpage('John Smith');
// BAD: studentA and studentB have different internal classes
```

For more details, see V8's [Design Elements: Fast Property Access](#).

Expensive Computations Should Be Distributed and Asynchronous

The LineRate Scripting is designed for high performance asynchronous I/O. Long-running computations block I/O while they are processing. Suppose you need to compute a bcrypt hash of a header field.

The most scalable solution is to compute the hash on a different system. One approach is to create a worker pool of servers that can perform expensive operations asynchronously. Note that the system will transmit the data to be hashed to the worker pool, so it is important that the connection to the worker pool is trusted. The following code distributes the header to a pool of hash computation servers:

```
var http = require('http');
var vsm = require('lrs/virtualServerModule');

// A virtual IP that is connected to a virtual server; the pool of
// real servers in the virtual server are all hash computation workers.
var hashPoolVirtualIp = '1.2.3.4';

vsm.on('exist', 'vs1', function (vs) {
  vs.on('request', function (req, resp, next) {
    var toHash = req.headers.fieldToHash;

    // GOOD: Hash is computed by the worker pool asynchronously.
    // Connection to worker pool must be trusted
    var hashReq = http.request({ hostname: hashPoolVirtualIp,
                                  path: '/computeHash' },
                                function (hashResp) {
                                  // Do something with hashResp.
                                  resp.writeHead(200, { 'Content-Type': 'text/plain' });
                                  resp.end('Hash computed');
                                  next();
                                });
    hashReq.setHeader('Content-Length', toHash.length);
    hashReq.end(toHash);
  });
});
```

When a request arrives, the proxy makes a new request to the worker pool. While the worker pool is computing the hash, the proxy continues to process other traffic. Also, the CPU cycles to compute the hash are consumed in the worker pool, rather than the proxy system, so there are more CPU cycles remaining in the proxy system to contribute to proxy throughput.

This approach is also more easily scalable. The size of the crypto worker pool can be increased simply by spinning up more pool members, and configuring real servers.

A less scalable approach is to compute the hash directly on the system. This approach will block the data path process; it will stop processing other traffic while the hash is being computed. As well, the hash computation consumes CPU cycles on the system. This leaves fewer CPU cycles for processing other traffic (including computing hashes for other requests), limiting the total throughput of the system.

```
var http = require('http');
var vsm = require('lrs/virtualServerModule');
var bcrypt = require('bcrypt-nodejs'); // Pure js from github:shaneGirish/bcrypt-nodejs

vsm.on('exist', 'vs1', function (vs) {
  vs.on('request', function (req, resp, next) {
    var toHash = req.headers.fieldToHash;

    // BAD: Hash is computed on the proxy system
    var hashed = bcrypt.hashSync(toHash, 'theSalt');

    resp.writeHead(200, { 'Content-Type': 'text/plain' });
    resp.end('Hash computed');
    next();
  });
});
```

```
});  
});
```

Script Errors

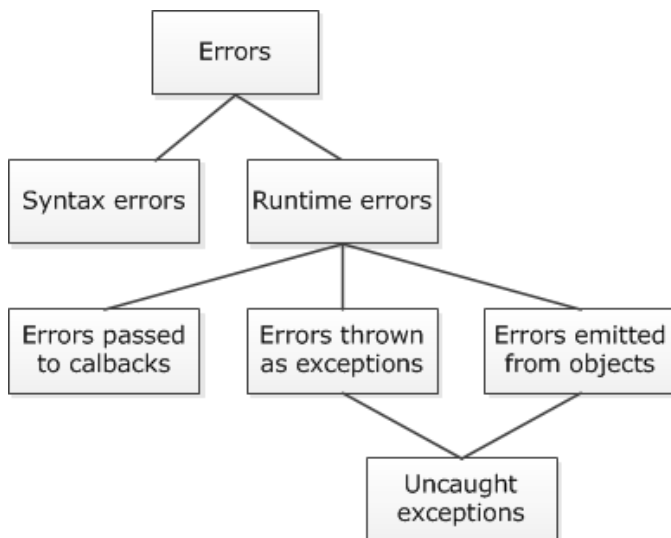
1. [Overview](#)
2. [Script Errors](#)
 - 2.1. [Syntax Errors](#)
 - 2.2. [Runtime Errors](#)
 - 2.2.1. [Errors Passed as Callbacks](#)
 - 2.2.2. [Errors Thrown as Exceptions](#)
 - 2.2.3. [Errors Emitted from Objects](#)
3. [Uncaught Exception Behavior](#)
4. [Error Sources](#)

Overview

Script errors can come from a variety of sources. Essentially all scripts may encounter errors. Even bug-free programs interact with external clients and servers over networks, and these interactions can fail. This section describes the kinds of errors and strategies for handling them.

Script Errors

The figure below shows the hierarchy of errors.



They are broadly divided into two classes:

- Syntax errors
- Runtime errors—Runtime errors are from one of three classes:
 - Errors passed to callback
 - Errors thrown as exceptions
 - Errors emitted from objects

If errors thrown as exceptions or errors emitted from objects are not handled, they are uncaught exceptions. Uncaught exceptions cause special behavior to be invoked, which is described later.

Syntax Errors

Syntax errors occur when the source code of the script cannot be parsed as valid JavaScript. Syntax errors are detected at configuration time. You can only fix them by changing the source code to be valid JavaScript.

Here is an example:

```
var vsm = require('lrs/virtualServerModule');
// BAD: Function names cannot have spaces
function has spaces() {
  console.log('has spaces was called');
}
```

When this script is configured, a warning is emitted:

```
WARNING: encountered the following errors when compiling script
foo:3: SyntaxError: Unexpected identifier
function has spaces() {
^^^^^^
```

Changes to script source applied, with warnings

Also, the script status shows **DOWN: syntax error** and contains the syntax error:

```
# show script foo
  Admin Status: offline
  Restart Mode: auto
  Created At: Mon Mar 17 16:10:50 2014 UTC
  Attached Entities:

Proxy Request Listeners
  <none>

Per-process Properties:
  Busy Timeouts
  Timeout Count Status
```



```
0s 15 / 15 <disabled/>
Status: DOWN: syntax error
```

Last Error

```
Timestamp: Mon Mar 17 16:11:32 2014 UTC
Stage: compile
Message:
encountered the following errors when compiling script
foo:3: SyntaxError: Unexpected identifier
function has spaces() {
^^^^^
```

The solution is to fix your syntax error. Auto-restart mechanisms are not useful for fixing syntax errors: the syntax is wrong and the script can never be executed correctly. For example, a script with correct syntax would be:

```
var vsm = require('lrs/virtualServerModule');
// GOOD: Function names have no spaces
function noSpaces() {
  console.log('noSpaces was called');
}
```

There are many references for the JavaScript language online. See the [Development Resources](#) for more information.

Runtime Errors

Scripts that compile without errors are still not necessarily bug free. Runtime errors occur when the program or script encounters an error while running. Runtime errors cannot be detected at configuration time. Runtime errors can occur when a script is first executed or when a callback is being executed, potentially long after a script was started.

Runtime errors are one of three kinds:

- Errors passed to callbacks
- Errors thrown as exceptions
- Errors emitted from objects

Errors thrown as exceptions are the only kind of runtime error defined in the JavaScript language itself; however, the other two error patterns are so common in Node.js that they are described here.

To determine which of these three methods is used, check the documentation and API reference for the node module you are using.

The effects of runtime errors can be lessened by auto-restart mechanisms built into LineRate. However, relying on auto-restart mechanisms has drawbacks. It is preferable to write scripts that handle runtime errors locally.

Errors Passed as Callbacks

Some asynchronous operations report errors by passing them as an argument to the callback. For example:

```
var fs = require('fs');
fs.rename('/tmp/myTempFile', '/tmp/myTempFile2', function (err) {
  if (err) {
    console.log('Rename generated an error passed to callback:', err);
  }
});
```

If `fs.rename()` encounters an error, it calls the callback. It is the callback's responsibility to handle the error. In the example above, the error is reported but is not handled in any other way.

If an error passed to a callback is not checked, it will be silently ignored. This is different than the other two kinds of runtime errors. Be cautious of this difference. It is up to the script to correctly detect the error and handle it. If the error is a serious problem for the script and it cannot be handled, the error should be thrown to invoke the failure and auto-restart mechanisms, as follows:

```
var fs = require('fs');
fs.rename('/tmp/myTempFile', '/tmp/myTempFile2', function (err) {
  if (err) {
    console.log('Critical rename error, script cannot handle:', err);
    throw err;
    // Invokes failure and auto-restart mechanisms.
  }
});
```

By throwing the error, the error is converted to the "error thrown as exception" type and therefore invokes the handling for those kinds of errors, described next.

It is a strong convention to pass the error as the first argument to the callback, however this is not universal. Check the documentation for the node module you are working with to confirm the error passing convention.

Errors Thrown as Exceptions

The JavaScript language allows exceptions to be thrown. Throwing exceptions are useful for synchronous operations such as math or parsing. For example:

```
// BAD: The argument to JSON.parse is not valid JSON.
var obj = JSON.parse('{ "key": "value", "key2": invalid value }');
```

When running this line, an exception is thrown. It will be shown in **show script**:

```
Admin Status: online
Restart Mode: manual
Created At: Mon Mar 17 16:10:50 2014 UTC
Attached Entities:
Proxy Request Listeners
```

```

<none>
Per-process Properties:
Busy Timeouts
Timeout Count Status
2s 15 / 15 <default/>
Status: DOWN: run-time error
Last Error
Timestamp: Mon Mar 17 16:52:01 2014 UTC
Stage: starting up
Message:
Uncaught exception while running script:
undefined:1: SyntaxError: Unexpected token '
{ "key": "value", "key2": invalid value }
^
SyntaxError: Unexpected token i
at Object.parse (native)
at foo:10:40

```

Errors thrown as exceptions are handled with try/catch blocks:

```

var obj;
try {
  obj = JSON.parse('{ "key": "value", "key2": invalid value }');
} catch (err) {
  console.log('Could not parse JSON into obj:', err);
}

```

The code that causes the exception to be thrown is called the "throw site." Sometimes the throw site is not the appropriate place to handle the error. In that case, it can be converted to one of the other error types. For instance, to convert a thrown exception to an error passed by callback, this structure could be used:

```

function tryParseJson(maybeJsonString, callback) {
  var obj;
  try {
    obj = JSON.parse(maybeJsonString);
  } catch (err) {
    // Encountered error parsing as JSON; pass it to the callback.
    callback(err, null);
    return;
  }
  // No error encountered; call callback without error.
  callback(null, obj);
}

```

Beware that the try/catch block must directly enclose the throw site. This is especially important with asynchronous operations. For example, the following code will not invoke the catch block at the end:

```

try {
  var resp = '';
  servReq.on('data', function (d) { resp += d; });
  servReq.on('end', function () {
    // BAD: Error thrown from here will not be caught by try/catch above
  });
}

```

```

    var json = JSON.parse(resp);
  });
} catch (err) {
  // BAD: This will not be reached for JSON.parse errors
}

```

Try/catch blocks cover only statements that are executed inside of them before they are completed. In the above example, the try/catch will catch errors encountered when executing the statements "var resp ="; "servReq.on('data', ...);" and "servReq.on('end', ...);". That is, it will catch errors encountered when defining the resp variable, and when registering listeners for the 'data' and 'end' events. It will not catch errors that are encountered while running these listeners. This may be more clear by considering this construction, which is equivalent:

```

function handleEndEvent(resp) {
  var json = JSON.parse(resp);
}
try {
  var resp = '';
  servReq.on('data', function (d) { resp += d; });
  servReq.on('end', handleEndEvent);
} catch (err) {
  // BAD: This will not be reached for JSON.parse errors
}

```

To actually handle the JSON.parse error, a try/catch must surround the scope from which JSON.parse is actually called. In this case, it must be inside the handleEndEvent:

```

function handleEndEvent(resp) {
  try {
    var json = JSON.parse(resp);
  } catch (err) {
    // GOOD: This is reached when JSON.parse has an error
  }
}
var resp = '';
servReq.on('data', function (d) { resp += d; });
servReq.on('end', handleEndEvent);

```

If an error is thrown but not caught, the [uncaughtException](#) handler will be invoked.

The JavaScript language allows any value to be thrown. However, it is strongly recommended that your code only throws error objects. Error objects include stack trace information and are easier to log, report and handle. For example:

```

var obj;
try {
  obj = JSON.parse(maybeJsonString);
} catch (err) {
  // BAD: Throwing a string instead of an Error.
  throw 'This string is not JSON: ' + maybeJsonString;
}
try {
  obj = JSON.parse(maybeJsonString);
} catch (err) {
  // GOOD: Throw an error object. Add extra info to object.
  var extraInfoError = new Error('Could not parse client request as JSON');
  extraInfoError.jsonErr = err;
  extraInfoError.invalidString = maybeJsonString;
  throw extraInfoError;
}

```

Errors Emitted from Objects

Errors can also be emitted from objects. Registering and unregistering for these events is the same as for other EventEmitters. There is one way in which the error event differs from other events. By default, if there is no listener for the error event, it is "bubbled up," either to a parent object or finally to the `uncaughtException` handler. For other events, by default if there is no listener the event is silently ignored.

Many of the Node.js core objects and especially Streams use the error event to report errors. Listen for an error emitted from an object by registering for the error event:

```
function onServerRequest(servReq, servResp, next) {
  // GOOD: Register for the error event on the servRespWritableStream
  servResp.on('error', function (err) {
    console.log('Error encountered with servResp:', err);
    // Client at the other end of servResp is gone; cleanup.
  });
  servResp.end('My response');
};
```

If a listener for an error event is not registered, and an error is emitted, the [uncaughtException](#) handler will be invoked.

If the error cannot be properly handled by the error event handler, it should be propagated to another handler (for instance, by emitting an error event from another object), or thrown to invoke the `uncaughtException` handler. For example:

```
function onServerRequest(servReq, servResp, next) {
  // GOOD: Register for the error event on the servRespWritableStream
  servResp.on('error', function (err) {
    console.error('Error encountered with servResp:', err);
    // GOOD: If the error cannot be handled, throw it again to cause failure
    // of the script; this is better than corrupted internal state.
    console.error('Incapable of handling this error, throwing');
    throw err;
  });
  servResp.end('My response');
};
```

Uncaught Exception Behavior

Uncaught exceptions refer to two kinds of runtime errors that are not handled by the script:

- Errors thrown as exceptions that are uncaught
- Errors emitted from objects that are unhandled

You can register a special uncaught exception handler to handle uncaught exceptions from scripts. If registered, the uncaught exception handler is invoked in these cases with the exception as an argument. If there is no uncaught exception handler registered, then the [default actions](#) occur. If the uncaught exception handler itself throws, then the default actions occur. The default actions are detailed in the

Process section of the Scripting API Reference Guide for your version is available from the following REST API node:

`https://<ip_address>:8443/doc/scripting/api.`

Generally, good programming practices require that the script rethrow the uncaught exception. If there is a non-fatal way to handle a particular error, it should be handled at the source by registering for the error event on the object that will emit the error.



Best practice: Record any extra information that may be suitable in debugging the program, and then rethrow the exception. Throwing from the uncaught exception handler causes the default actions to be invoked.

Error Sources

Errors can come from a variety of sources. It is impossible to document them all here, as they include errors that may be generated by third-party modules.



Best Practice: Refer to the API documentation while using new methods. Every operation that involves an external service or client could fail. Many internal operations can fail as well. Ensure your script has appropriate handling for error cases.

Some common sources of errors that scripts should be prepared for:

- Errors reading from or writing to streams, such as the HTTP ServerRequest, ServerResponse, ClientRequest and ClientResponse streams. These are emitted as error events; see the API documentation.
- Errors dealing with stale configuration objects, for instance deconfiguring virtual servers or forward proxies while a script is running. See [Diagnosing Strange Script Behavior](#).

Testing, Debugging, and Troubleshooting Scripts

1. [Overview](#)
2. [Syntax Error Messages](#)
3. [Runtime Errors](#)
4. [Uncaught Exceptions](#)
5. [Logging with Console.log](#)
6. [Diagnosing Memory Usage](#)
7. [Writing Past Content-Length](#)
8. [Errors from the zlib Module](#)

Overview

You can use a few different methods to test, debug, and troubleshoot scripts.

Syntax Error Messages

When a script's status is set to online, the internal compiler compiles the code. If the compiler finds errors in syntax, the errors display immediately in the CLI window. The syntax error message lists the line number where the compiler stopped and provide additional information about the error.

Below are examples of syntax errors and what they mean.

The examples include the surrounding configuration to show which lines are addressed by the compiler:

- Green lines are LineRate configuration commands.
- Blue is scripting code
- Red is scripting code that contains errors.

```
script example_syntax_error
```

```
source inline "ENDSOURCE_EXAMPLE"  
  var assert = require('assert');  
  var os = require('os');  
  var timeout = 5000; // ms  
  var this_syntax == errorful;  
ENDSOURCE_EXAMPLE  
admin-status online  
!
```

This produces an error that looks like this:

```
!
  WARNING: encountered the following errors when compiling script
  example_syntax_error:4: SyntaxError: Unexpected token =
  var this_syntax = = errorful;
  ^
```

This tells you that in the script named `example_syntax_error`, line 4, contained a syntax error. The second `=` sign is a syntax error. The `^` symbol below the error message points out the exact character at which the code stops making sense; this is the `Unexpected token =` referred to in the error message. Note that line 4 is the fourth line after the `source inline` command.

The compiler does not know or care about the surrounding configuration, only the bits between the start tag `ENDSOURCE_EXAMPLE` and end tag `ENDSOURCE_EXAMPLE`. If you leave out the double quotes around the beginning script tag, they will be added when you view the script with `show running-config`.

Runtime Errors

Scripts that compile without errors are still not necessarily bug free. Syntax errors will halt compilation, but other errors may be discovered only when the script is running. These errors are called runtime errors. In the context of the LineRate Scripting, these will only manifest when an online script attempts to perform an action that the system cannot execute.

Runtime errors include both caught and uncaught exceptions. Using the `try` and `catch` commands, it is possible to intercept and gracefully handle certain errors. Errors that aren't checked for will show up as `uncaughtExceptions`; these will halt execution of the scripting engine unless they are handled using `process.on(uncaughtException)`.

```
script example_syntax_error
  source inline "ENDSOURCE_EXAMPLE"
  var assert = require('assert');
  var os = require('os');
  var timeout = 5000; // ms
  something_not_defined();
ENDSOURCE_EXAMPLE
admin-status online
!
```

```
WARNING: Uncaught exception while running script:example_syntax_error:6:
ReferenceError: something_not_defined is not defined
something_not_defined();
^
ReferenceError: something_not_defined is not defined
at example_syntax_error:6:3
```


The script has a reference to `something_not_defined`, but this is not defined anywhere in the script or in the included libraries. In this case, the scripting engine knows that the problem is with the undefined reference at line 6, character 3 of the script.

Uncaught Exceptions

You can register a special uncaught exception handler to handle uncaught exceptions from scripts:

- Runtime errors, such as referring to a variable that is not defined or using a property that is not defined.
- An error from an asynchronous operation, for example an HTTP client disconnecting while a script was trying to write a response body to the client.
- An exception that is thrown but not caught by the script, for example calling `JSON.parse()` on a string that is not valid JSON and not surrounding the call to `JSON.parse()` with a try/catch block.

If registered, the uncaught exception handler is invoked in these cases with the exception as an argument. If there is no uncaught exception handler registered, then the default actions occur. If the uncaught exception handler itself throws, then the default actions occur. The default actions are detailed in the Process section of the Scripting API Reference Guide for your version is available from the following REST API node:

`https://<ip_address>:8443/doc/scripting/api.`

Generally, good programming practices require that the script re-throw the uncaught exception. If there is a non-fatal way to handle a particular error, it should be handled at the source by registering for the `'error'` event on the object that will emit the error.



Best practice: Record any extra information that may be suitable in debugging the program, and then re-throw the exception. Throwing from the uncaught exception handler causes the default actions to be invoked.

Logging with Console.log

When scripts execute, the system writes certain data to log files. In addition, you can add specific logging commands to your scripts.



Tip: Be careful when logging using live traffic. The log files can grow very large very fast. Use logging when testing, then remove logging.

The `console.log` method, provided in the node.js libraries, lets you view add data to the system log:

```
console.log('PID: ' + process.pid + ' request: ' + request.url);
```

This example calls `console.log` to log information for each request the script processes. The log file will contain the string "PID:" followed by the process ID of the process which handled the request, the string "request:", and the URL of the requested page. Note that this line logs each request that comes through the forward proxy.

You can the `tail` command to follow the end of log while executing the script. Open another terminal or command window, SSH into the system, then use these commands while the script executes:

```
example-host# bash
*****WARNING*****
The bash prompt allows unrestricted access to the system.
It is possible to configure the system in ways that cannot
be shown in the UI, that may lead to incorrect operation
of the system, and that may or may not be persistent after a
reload of the system. LineRate Systems recommends that no
configuration be made from bash unless directed by support.
*****
[admin@example-host ~]$ sudo tail -f /var/logs/controller.messages
```

While in bash, you can also use other standard UNIX commands, such as the `grep` command to search for strings in the `controller.messages`. Be aware, however, of the warning regarding bash commands that displays when you go into bash.

Diagnosing Memory Usage

Use

Scripts may have bugs that cause objects to be retained longer than needed. These retained objects use memory on the system, which can affect performance. You can inspect the heap objects that are retained by a script to determine if all of the retained objects should be retained, or if a bug has caused some objects to be retained too long.

One tool to inspect the Javascript heap is the Google Chrome heap profiler, which is available in any Google Chrome browser.

For more information about the Chrome Heap Profiler, refer to:

<https://developers.google.com/chrome-developer-tools/docs/heap-profiling>

A system with Google Chrome Developer Tools is required, this is the profiler system. The profiler system can run any operating system for which Chrome is available.



How to get a heap snapshot from the LineRate system and load it into the heap profiler:

1. On the LineRate system, set up a script to inspect.

- This is an example script that intentionally leaks objects called LeakedObjects to make it easy to examine the heap snapshots.

```
host-22(config)# script testscript
host-22(config-script:testscript)# source inline END
Enter a script source followed by 'END' on a line by itself.
function LeakedObject() {
this.birthdate = Date();
};
var leakedObjs = [];
setInterval(function () {
var newObj = new LeakedObject();
leakedObjs.push(newObj);
}, 1000);
END
host-22(config-script:testscript)# admin-status online
host-22(config-script:testscript)# show script testscript
Admin Status: online
Created At: Wed Feb 6 22:16:51 2013 UTC
Attached Entities:
Proxy Request Listener <none>
Status: UP
Last Runtime Error <none/>
```

2. On the LineRate system, use the CLI to enable heap snapshotting.

- This causes a heap snapshot to be taken at regular intervals and writes it to a file on the system's disk.

```
# conf
# debug proxy js-heap-profile 60
```

- Each core will write a heap snapshot file to this path every 60 seconds:

```
/home/linerate/data/profile/
lb_http.v8heap.<pid>.<time>.heapsnapshot</time></pid>
```

Where <pid> is the process ID of the process on each core and <time> is a counter that increases by 1 every second.</time></pid>

3. You can observe new heap snapshots being written.

```
# bash
[user@proxy ~]$ ls /home/linerate/data/profile
lb_http.v8heap.1508.1360188010.045441.heapsnapshot
lb_http.v8heap.1790.1360188100.046223.heapsnapshot
lb_http.v8heap.1508.1360188040.050035.heapsnapshot
lb_http.v8heap.1790.1360188130.045817.heapsnapshot
lb_http.v8heap.1508.1360188070.045629.heapsnapshot
lb_http.v8heap.1791.1360188010.047441.heapsnapshot
```

```

lb_http.v8heap.1508.1360188100.046223.heapsnapshot
lb_http.v8heap.1791.1360188040.048035.heapsnapshot
lb_http.v8heap.1508.1360188130.045817.heapsnapshot
lb_http.v8heap.1791.1360188070.047629.heapsnapshot
lb_http.v8heap.1789.1360188010.047441.heapsnapshot
lb_http.v8heap.1791.1360188100.048223.heapsnapshot
lb_http.v8heap.1789.1360188040.048035.heapsnapshot
lb_http.v8heap.1791.1360188130.047817.heapsnapshot
lb_http.v8heap.1789.1360188070.047629.heapsnapshot
lb_http.v8heap.1792.1360188010.047441.heapsnapshot
lb_http.v8heap.1789.1360188100.048223.heapsnapshot
lb_http.v8heap.1792.1360188040.048035.heapsnapshot
lb_http.v8heap.1789.1360188130.047817.heapsnapshot
lb_http.v8heap.1792.1360188070.047629.heapsnapshot
lb_http.v8heap.1790.1360188010.045441.heapsnapshot
lb_http.v8heap.1792.1360188100.048223.heapsnapshot
lb_http.v8heap.1790.1360188040.046035.heapsnapshot
lb_http.v8heap.1792.1360188130.047817.heapsnapshot
lb_http.v8heap.1790.1360188070.045629.heapsnapshot
[user@proxy ~]$ exit
#

```

4. Execute any scripts that may cause leaks.
5. Wait for one more snapshot to be written, then stop collecting profiles.

```

# config
# no debug proxy js-heap-profile 60

```

6. Copy some or all of the heap snapshots to the profiler system.

```

# bash
[user@proxy ~]$ scp /home/linerate/data/profile/*heapsnapshot user@profile-
machine
Password: [password on profile-machine]

```

7. Open Google Chrome on the profiler system.
 - The specific instructions for loading a heap profile into Chrome may be superseded in other versions of Chrome; the rest of these instructions are for Chrome 24.0. Open Developer Tools and click **Profiles**. Right-click beneath Profiles on the left-hand side panel, and click **Load Heap Snapshot**. Open one of the heap snapshots.

Object	Distance	Objects Count	Shallow Size	Retained Size
Object	2	2542 2%	87982 0%	12641904 10%
Unscripted code	4	23932 15%	6807904 16%	11186776 14%
(script)	4	22760 14%	9734896 18%	9734896 18%
(script)	3	19118 12%	1376486 5%	8278392 12%
(script)	3	17470 11%	5224608 12%	5339548 11%
(script)	4	43270 27%	1486544 6%	5250560 21%
Buffer	4	25 0%	600 0%	800200 3%
Array	3	1383 1%	42456 0%	607320 2%
LeakedObject	4	1363 1%	37208 0%	483432 2%
LeakedObject 0245675	4		32 0%	416 0%
LeakedObject 0245677	4		32 0%	416 0%
LeakedObject 0245678	4		32 0%	416 0%
LeakedObject 0245681	4		32 0%	416 0%
LeakedObject 0245683	4		32 0%	416 0%
LeakedObject 0245685	4		32 0%	416 0%
LeakedObject 0245687	4		32 0%	416 0%
LeakedObject 0245689	4		32 0%	416 0%
LeakedObject 0245691	4		32 0%	416 0%
LeakedObject 0245693	4		32 0%	416 0%
LeakedObject 0245695	4		32 0%	416 0%
LeakedObject 0245697	4		32 0%	416 0%
LeakedObject 0245699	4		32 0%	416 0%
LeakedObject 0245701	4		32 0%	416 0%
LeakedObject 0245703	4		32 0%	416 0%
LeakedObject 0245705	4		32 0%	416 0%
LeakedObject 0245707	4		32 0%	416 0%
LeakedObject 0245709	4		32 0%	416 0%
LeakedObject 0245711	4		32 0%	416 0%
LeakedObject 0245713	4		32 0%	416 0%
LeakedObject 0245715	4		32 0%	416 0%

8. Observe all the retained objects in the heap and information on how much memory they are using.
 - Many web resources on debugging memory leaks in Javascript and node.js can provide tips on interpreting these heaps and fixing script bugs.

Writing Past Content-Length

The HTTP protocol marks the boundaries between consecutive requests and responses using one of several methods, such as connection close, content length, and chunked encoding (short names for behaviors described in detail in [RFC2616](#)). By default, LineRate Scripting writes responses that are delimited by chunked encoding when possible, falling back to connection close otherwise.

However, if the script manually sets the Content-Length header, LineRate Scripting will use that header to mark the request/response boundary. In this situation, both LineRate Scripting and Node.js will not chunk the body, and both will send the Content-Length header as specified by the script. However, there is one way in which the behavior differs: In Node.js, a script is allowed to write a body that is larger than the limit specified by the Content-Length header, and Node.js will not take any action to prevent it.

This behavior can cause difficult-to-diagnose bugs. It is also a security risk in a proxy environment: a badly behaved script could overwrite Content-Length such that the overwrite appears to be a valid HTTP request or response, and cause the origin server or client to mismatch requests and responses. To prevent these issues, LineRate Scripting does prevent writing past Content-Length. However, to maximize API compatibility with Node.js, LineRate Scripting does not throw an error at the point where the extra write is attempted. In LineRate Scripting, data that is written in excess of the Content-Length limit is dropped.

If a script is setting the Content-Length header in requests or responses that it is generating, it should be careful to not exceed the Content-Length limit. For many scripts, this is easy to do: the body is

accumulated in a string or buffer, and that string or buffer is sent all at once. Here is an example that works for strings with single-byte characters:

```
var util = require('util');

var totalRequestCount = 0;
function onRequest(servReq, servResp, cliReq) {
  totalRequestCount += 1;
  if (servReq.method === 'GET' && servReq.url === '/debugStats') {
    var responseBody = util.format('The time is %s. Process %d has served %d requests',
                                   new Date().toISOString(), process.pid, totalRequestCount);
    servResp.writeHead(200, { 'Content-Type': 'text/plain',
                              'Content-Length': responseBody.length });
    servResp.end(responseBody);
  }
}
```

For strings with multibyte characters, accumulate in a Buffer and use Buffer.byteLength().

In more complicated cases, where the body is not fully accumulated before sending out, it is often easier and more robust to use chunked encoding. In many cases, this improves client performance as well. However, if chunked encoding is still not suitable, you can use a helper to generate an error when writing more than Content-Length is attempted. One example of a helper is shown below:

```
function checkedWrite(data, arg1, arg2) {
  var encoding, cb, cl;

  cl = this.getHeader('content-length');
  if (isNaN(cl)) {
    return this.write.apply(this, arguments);
  }

  // parse arguments (from node's net.js)
  if (arg1) {
    if (typeof arg1 === 'string') {
      encoding = arg1;
      cb = arg2;
    } else if (typeof arg1 === 'function') {
      cb = arg1;
    } else {
      throw new Error('bad arg');
    }
  }

  encoding = (encoding || 'utf8').toLowerCase();
  data = new Buffer(data, encoding);

  if (data.byteLength() + this.bytesWritten > cl) {
    this.emit('error', new Error('Writing past content-length (' + cl + ')'));
    // Another option: throw new Error('Writing past content-length (' + cl + ')');
  } else {
    return this.write(data, cb);
  }
}
```

Errors from the zlib Module

A few errors can bubble up from the zlib module. Use the table below to troubleshoot the most common errors.

Error	Meaning	Possible causes	What to do
{ [Error: invalid block type] errno: -3, code: 'Z_DATA_ERROR' }	Zlib could not understand the compressed data.	The most likely cause is using a decompression method requiring headers.	The 'inflateRaw' compression method will not prefix compressed data with headers. Use 'inflateRaw' and not 'unzip', 'gunzip', or 'inflate'.
{ [Error: incorrect header check] errno: -3, code: 'Z_DATA_ERROR' }	Zlib could not understand or parse the compression headers.	The most likely cause is using the wrong inflation type for the compression style.	Be sure to decompress with the valid object/method: <ul style="list-style-type: none">• For 'deflate' compressed streams, the only valid decompression methods are 'inflate' and 'unzip'.• For 'gzip' compressed streams, the only valid decompression methods are 'gunzip' and 'unzip'.• For 'deflateRaw' compressed streams, the only valid decompression method is 'inflateRaw'.
{ [Error: invalid distance too far back] errno: -3, code: 'Z_DATA_ERROR' }	Zlib could not allocate a large enough window for decompression.	If a compressed stream with a larger window size is given as input, decompression will fail rather than trying to allocate a larger window.	The 'windowBits' configuration parameter for decompression must be greater than or equal to the value used in compression.
{ [Error: invalid code lengths set] errno: -3, code: 'Z_DATA_ERROR' }	Zlib encountered a corrupted stream.	A raw decompression ('inflateRaw') was attempted on an invalid stream; most likely already inflated data.	The stream is very likely already decompressed. Do not attempt inflation or verify compression side is properly deflating data.
RangeError: length > kMaxLength	An internal LineRate limitation has been exceeded.	LineRate limits buffers to a size of 1073741823 bytes or less.	Do not use a 'chunkSize' value that exceeds buffer size limit: 1073741823 bytes.
{[Error: Zlib Error] errno: -2, code: 'Z_STREAM_ERROR' }	A Zlib error occurred without the system library	Multiple reasons exist for this, with the most probable being a required dictionary not being set. See the libz	Set the required dictionary via the Zlib opts.dictionary field.

Error	Meaning	Possible causes	What to do
	setting a human readable message.	inflateSetDictionary documentation.	
{[Error: invalid window size] errno: -3, code: 'Z_DATA_ERROR' }	A Zlib error occurred in parsing headers for decompression. The inflate algorithm detected a mismatch in the window size configured in the stream and the compressed data.	This error can be seen when using the deflate/inflate compression/decompression pair and the minimum 'windowBits' configuration size of 8. In this case, using a 'windowBits' value of 8 to deflate, the inflate and unzip streams fail to parse and match the zlib headers. This is an internal bug to the libz system utility.	When using a deflate stream use a 'windowBits' value of 9 - 15. When using an inflate stream use an incrementally increasing 'windowBits' value until decompression succeeds.

Diagnosing Strange Script Behavior

1. [Overview](#)
 2. [Common Configuration for Examples](#)
 3. [Examples](#)
-

Overview

If you are not familiar with asynchronous programming or multiprocess programming, you may be surprised by some behaviors. The LineRate Scripting runs independently on multiple processor cores to provide high performance. Much of the script code is in [callbacks](#). Callbacks are only invoked when certain events occur.

Each section in this page describes a common behavior that script programmers may encounter. It helps you identify the issue based on the behavior you are experiencing, understand the cause, and implement fixes.

Common Configuration for Examples

The examples in this section use a common configuration to run each script. The commands to add the common configuration are shown below. Replace <IPADDR> with an IP address that is configured on the system that you can reach from a web browser and is suitable for debugging. You can see a list of configured IPs using the **show interface** command.

```
config

virtual-ip vip1 ip <IPADDR> 8080

admin-status online

service http

virtual-server vs1

service http

attach virtual-ip vip1 default
```

Examples

- [A Variable Is Unset That I Know I Set](#)
- [External Services Only Work Intermittently from a Script](#)
- [My Compression Streams Never Complete](#)
- [My Request Handler Is Only Called Sometimes](#)
- [My Script Can Only Forward New Requests Sometimes](#)
- [User Sessions Are Spuriously Deleted](#)

A Variable Is Unset That I Know I Set

1. [Overview](#)
 2. [An Example of the Problem](#)
 3. [What's Going On?](#)
 4. [Diagnosing the Problem](#)
 5. [Handling the Problem](#)
 - 5.1. [Occasionally Push Updates to an Aggregator](#)
 - 5.2. [Store Shared Data in an External Database or Datastore](#)
-

Overview

Variables are set only in the process that does the setting. It is possible that the variable is being set in one process, but another process is trying to read it. The reading process will read its local instance of the variable and find it unset.

There are methods to share information between script processes, but these methods can impact performance. Continue reading to understand the issue in more detail and learn approaches for sharing information between processes.

An Example of the Problem

The following example script tries to record the last request received and the total number of requests. Every time a new request is received, the script responds with a page that shows the current request, the last request, and the total request count.

```
var vsm = require('lrs/virtualServerModule');

// These variables are global to the entire instance of the script running in
// this process, but they are not shared across other processes.
var requestCount = 0;
var lastRequestUrl = undefined;

// This function doesn't do anything except for spin for 2 seconds.
function busyLoop() {
  var start = Date.now();
  while (Date.now() < start + 2000) {}
}

function respond(req, res, next) {
  if (req.url === '/favicon.ico') { next(); return; }
  var response = 'You requested: ' + req.url + '\n' +
    'Last request: ' + lastRequestUrl + '\n' +
    'Served by PID ' + process.pid + '\n' +
    'Served ' + requestCount + ' requests.\n';
```

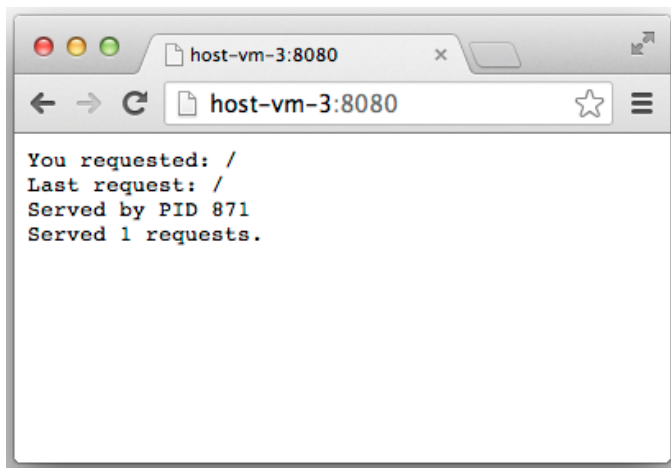
```

res.setHeader('Content-type', 'text/plain');
res.setHeader('Content-length', response.length);
res.end(response);
next();
++requestCount; // Adds 1 to this process' totalRequestCount
lastRequestUrl = req.url;
busyLoop();
}

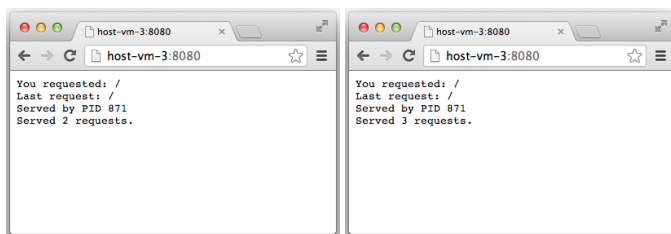
vsm.on('exist', 'vs1', function(vs) {
  vs.on('request', respond);
});

```

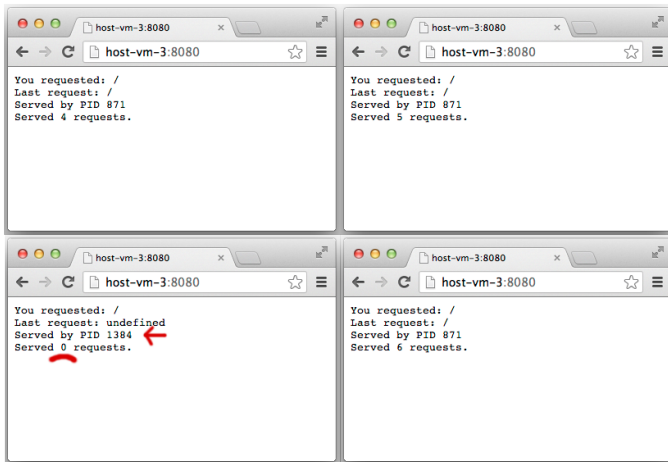
If you install this script and try to load the page, you will see that it takes 2 seconds to return a result like the following:



Now, if you reload a couple times you may see the totalRequestCount increment by one each time:

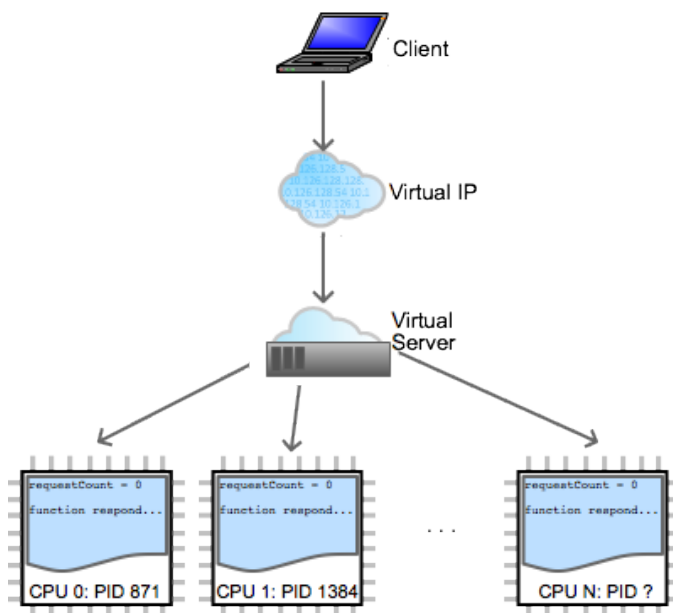


However, if you reload from several browser windows at once or reload in the middle of loading a page, you may see totalRequestCount reset to an earlier number:

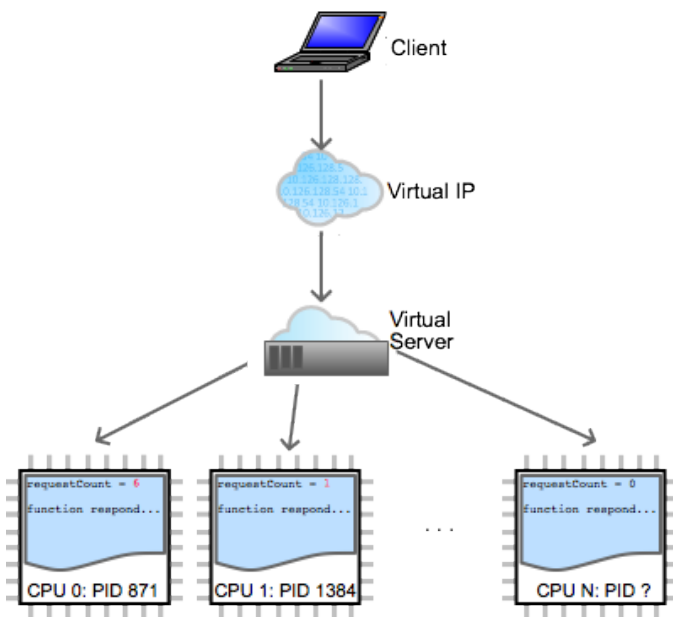


What's Going On?

Several processes are all simultaneously trying to grab the next incoming request. One of the processes will win this race, get the connection, and invoke the script handler. But which process wins is not deterministic.



In this example, the process identified by PID 871 happens to get the first several requests. Each time it gets a request, it runs the `respond` function. This function updates `totalRequestCount` and `lastRequestedUrl` in the context of the demo script in PID 871. After completing all the requests shown above, the situation looks like:



In PID 871, the demo script's context has a variable `requestCount = 6`. In PID 1384, the demo script's context has a variable `requestCount = 1`. The PIDs run independently, and don't share the same `requestCount`.

Diagnosing the Problem

To determine if this is causing unexpected behavior in your script, try looking for places where a variable saves state from one request to the next. When you update that variable, try reporting the update. For instance, look at the new logging on line 24 below:

```
var vsm = require('lrs/virtualServerModule');

// These variables are global to the entire instance of the script running in
// this process, but they are not shared across other processes.
var requestCount = 0;
var lastRequestUrl = undefined;

// This function doesn't do anything except for spin for 2 seconds.
function busyLoop() {
  var start = Date.now();
  while (Date.now() < start + 2000) {}
}

function respond(req, res, next) {
  if (req.url === '/favicon.ico') { next(); return; }
  var response = 'You requested: ' + req.url + '\n' +
    'Last request: ' + lastRequestUrl + '\n' +
    'Served by PID ' + process.pid + '\n' +
    'Served ' + requestCount + ' requests.\n';
  res.setHeader('Content-type', 'text/plain');
  res.setHeader('Content-length', response.length);
  res.end(response);
  next();
  // NEW: Report where we are updating requestCount.
```

```

    console.log('Increasing requestCount from ' + requestCount + ' to '
                (requestCount + 1) + ' in ' + process.pid);
    ++requestCount; // Adds 1 to this process' requestCount
    lastRequestUrl = req.url;
    busyLoop();
}

vsm.on('exist', 'vsl', function(vs) {
    vs.on('request', respond);
});

```

When this script runs, instead of only updating the variable, it will report the PID from which it is updating the variable, and its new value. Now, it is obvious that each PID has an independent requestCount:

```

LROS# bash "tail /var/log/controller.messages"
LROS: Increasing requestCount from 1 to 2 in 871
LROS: Increasing requestCount from 2 to 3 in 871
LROS: Increasing requestCount from 3 to 4 in 871
LROS: Increasing requestCount from 4 to 5 in 871
LROS: Increasing requestCount from 0 to 1 in 1384
LROS: Increasing requestCount from 5 to 6 in 871

```

Handling the Problem

If the variable is aggregated later, like a statistic, it may not be necessary to share it among all the processes; it may be sufficient to occasionally push it to an aggregator. If the variable really must be shared among all the processes, you can use a database.

Occasionally Push Updates to an Aggregator

First, consider if it is necessary to share this information across processes. In the example above, perhaps `requestCount` is being collected for statistics or billing. It may be sufficient to aggregate in the statistics or billing system, rather than in the LineRate Scripting. The LineRate Scripting can push significant updates to `requestCount` to this external system. You can modify the example to have a new method `pushSignificantUpdates` that will push every time `requestCount` increases by one hundred:

```

var vsm = require('lrs/virtualServerModule')
    , http = require('http');

// These variables are global to the entire instance of the script running in
// this process, but they are not shared across other processes.
var requestCount = 0;
var lastRequestUrl = undefined;

// NEW: Pushes updates when requestCount increases by 100
var updateInProgress = false;
function pushSignificantUpdates() {

```

```

if (requestCount % 100 !== 0 || updateInProgress) return;
updateInProgress = true;
var updateString = JSON.stringify(requestCount);
var updateReq = http.request(
  { 'host': 'stats.internal.com',
    'path': '/stats/proxy/' + process.pid + '/requestCount',
    'headers' : { 'Content-Type' : 'application/json' ,
                  'Content-Length' : updateString.length },
    'method': 'PUT' },
  function reqComplete() {
    updateInProgress = false;
  });
updateReq.end(updateString);
}

// This function doesn't do anything except for spin for 2 seconds.
function busyLoop() {
  var start = Date.now();
  while (Date.now() < start + 2000) {}
}

function respond(req, res, next) {
  if (req.url === '/favicon.ico') { next(); return; }
  var response = 'You requested: ' + req.url + '\n' +
    'Last request: ' + lastRequestUrl + '\n' +
    'Served by PID ' + process.pid + '\n' +
    'Served ' + requestCount + ' requests.\n';
  res.setHeader('Content-type', 'text/plain');
  res.setHeader('Content-length', response.length);
  res.end(response);
  next();
  ++requestCount; // Adds 1 to this process' requestCount
  lastRequestUrl = req.url;
  // NEW: Update stats server if necessary
  pushSignificantUpdates();
  busyLoop();
}

vsm.on('exist', 'vs1', function(vs) {
  vs.on('request', respond);
});

```

With an appropriate interval for push updates, this approach can provide higher performance, too. The processes can run independently, handling requests and keeping information in their local memory. Using this local memory is fast and doesn't require coordination among the processes. The data is pushed to an aggregator only at specified intervals or thresholds (every 100 requests, in this example).

Store Shared Data in an External Database or Datastore

Shared data can be stored in an external database or datastore. The LineRate Scripting supports the [redis](#) key/value datastore and the [VoltDB](#) NoSQL database.

External Services Only Work Intermittently from a Script

1. [Overview](#)
 2. [An Example of the Problem](#)
 3. [Diagnosing the Problem](#)
 4. [Handling the Problem](#)
 - 4.1. [Designing Robust Scripts to Handle Errors](#)
 - 4.2. [External Services Need to Handle Many Simultaneous Clients](#)
-

Overview

Each process executes the script independently. If the script connects to an external service, like a database, then one connection will be made from each process. From the perspective of the external service, each connection is from an independent client.

Each connection can have its own problems, like connection problems, authentication problems, or protocol problems. Continue reading to understand the issue in more detail and learn approaches for robust use of external services.

An Example of the Problem

The following script connects to a VoltDB database and waits for incoming requests. Whenever a request is for the path `/vote?candidate=1&phone=18888827535`, `/vote?candidate=2&phone=18888827536`, and so on, the VoltDB stored procedure `Vote` is executed.

For details on the `Vote` procedure, see the [Vote Example](#) from VoltDB. However, for the purposes of understanding script behavior, all that you need to know is that `voltClient.callProcedure(vote, ...)` is an operation using an external service that can fail.

```
var vsm = require('lrs/virtualServerModule')
, urlMod = require('url')
, VoltClient = require('voltjs/lib/client')
, VoltConstants = require('voltjs/lib/voltconstants')
, VoltConfiguration = require('voltjs/lib/configuration')
, VoltProcedure = require('voltjs/lib/query');

var voteProc = new VoltProcedure('Vote', ['long', 'int', 'long']);
function tallyVote(phoneNum, candidate) {
  var query = voteProc.getQuery();
  query.setParameters([ phoneNum, candidate, 10 /* maxVotes */]);
  voltConn.callProcedure(voteProc, function handleResults(error, event, res) {
    if (errorCode !== VoltConstants.STATUS_CODES.SUCCESS) {
      // BAD: No code to repair the connection.
    }
  });
}
```

```

        console.log('Error recording vote:', errorCode);
    }
    });
}

var voltConn = undefined;
function connectToVolt() {
    var voltConfig = new VoltConfiguration();
    voltConfig.host = '192.168.1.1'; // VoltDB server
    client = new VoltClient(voltConfig);
    client.connect(function connected(results) {
        console.log('Process ' + process.pid + ' connected to VoltDB');
        voltConn = client;
    }, function loginError(results) {
        console.log('Process ' + process.pid + ' VoltDB login:', results);
    });
}

vsm.on('exist', 'vsl', function(vs) {
    vs.on('request', function handleRequest(req, res, next) {
        var url = urlMod.parse(req.url);
        if (url.pathname === '/vote') {
            tallyVote(url.query.candidate, url.query.phone);
        }
        next();
    });
});

connectToVolt();

```

Suppose the script runs successfully for a while across all processes, accumulating votes. But then, for some reason, the connection between the script in PID 1384 and the VoltDB server has a [STATUS_CODES.CONNECTION_LOST](#) error. The scripts in the rest of the PIDs do not encounter this error. For whatever reason, the problem was only encountered on that one connection. From then on, all future votes that arrive in PID 1384 will also result in an error, because the connection is lost.

```

LROS: Error recording vote: -4
LROS: Error recording vote: -4
LROS: Error recording vote: -4

```

Diagnosing the Problem

To determine if this is your problem, ensure that error logging messages include the PID. If a service is broken in one process, look back through the logging history to see when that process first encountered trouble. Look for the symptom that the service is consistently broken in some of the PIDs, but functioning correctly in others.

Try restarting the script in all processes. If the problem goes away temporarily, or reappears on another process, it may be that the script's handling of external services is not robust, or the external service is behaving badly.

On the service side, check for issues that may affect one connection but not others, such as:

- Service-wide connection limits—The service only allows N simultaneous connections.
- Authorization limits—The script user can only have N simultaneous connections to the service.

- Firewalls—The firewall interprets multiple connections from the LineRate as an attack and blocks all of them except for the first N.

Handling the Problem

Robust scripts must be able to handle errors from external services. External services need to be able to handle many clients at once.

Designing Robust Scripts to Handle Errors

A script that is robust enough to handle errors from external services will check for valid connections to external services before using them, check for errors when using the service, and repair broken connections.

Below is an enhanced example. If an error is encountered, the connection to the server is replaced by a new one. Scripting is asynchronous, so new requests can arrive while the connection to the server is being replaced. While it is being replaced, errors are returned to the client. A further enhancement could buffer these votes up to some limit.

```
var vsm = require('lrs/virtualServerModule')
, urlMod = require('url')
, VoltClient = require('voltjs/lib/client')
, VoltConstants = require('voltjs/lib/voltconstants')
, VoltConfiguration = require('voltjs/lib/configuration')
, VoltProcedure = require('voltjs/lib/query');

var voteProc = new VoltProcedure('Vote', ['long', 'int', 'long']);
function tallyVote(phoneNum, candidate) {
  var query = voteProc.getQuery();
  query.setParameters([ phoneNum, candidate, 10 /* maxVotes */]);
  voltConn.callProcedure(voteProc, function handleResults(error, event, res) {
    if (errorCode !== VoltConstants.STATUS_CODES.SUCCESS) {
      // GOOD: Throw away connection on error, and force reconnection.
      voltConn.close();
      voltConn = undefined;
      ensureConnectedToVolt();
      console.log('Error, vote for ' + candidate + ' from ' + phoneNum +
        ' not recorded. VoltDB returned:', errorCode);
    }
  });
}

var voltConn = undefined;
var voltConnectionInProgress = false;
function ensureConnectedToVolt() {
  // GOOD: Reconnect if required, guarantee only one reconnect at a time.
  if (voltConn || voltConnectionInProgress) return;
  voltConnectionInProgress = true;
  var voltConfig = new VoltConfiguration();
  voltConfig.host = '192.168.1.1'; // VoltDB server
  client = new VoltClient(voltConfig);
  client.connect(function connected(results) {
    console.log('Process ' + process.pid + ' connected to VoltDB');
    voltConnectionInProgress = false;
    voltConn = client;
  }, function loginError(results) {
```

```

    console.log('Process ' + process.pid + ' VoltDB login:', results);
    voltConnectionInProgress = false;
    setTimeout(ensureConnectedToVolt, 5000);
  });
}

vsm.on('exist', 'vs1', function(vs) {
  vs.on('request', function handleRequest(req, res, next) {
    var url = urlMod.parse(req.url);
    if (url.pathname === '/vote') {
      // GOOD: Check for external service and return an error if broken.
      if (voltConn) {
        tallyVote(url.query.candidate, url.query.phone);
      } else {
        var errorString = 'Failed to record vote';
        res.writeHead(502, { 'Content-type' : 'text/plain',
                           'Content-length': errorString.length });
        res.end(errorString);
      }
    }
    next();
  });
});

ensureConnectedToVolt();

```

Scripts are more robust if they:

- Prepare for connections to the external service to fail during the lifetime of the script.
- Have strategies to reconnect to external services if necessary.
- Handle traffic when the external service is unavailable (for instance, traffic that arrives during the time that a reconnection is in progress).
- Check the results of using the external service for failure.

In the modified example, `handleRequest` and `tallyVote` are prepared for problems with the connection to VoltDB. `handleRequest` will check if it is available before calling `tallyVote` and will return an error if it is not. `tallyVote` checks the result of its call, and if it has an error it closes the connection and removes the reference to it (stored in `voltConn`), and starts the process to reconnect.

While reconnection is in progress, the `voltConnectionInProgress` variable is true, to prevent other error handlers from starting a parallel connection attempt. If the connection attempt fails, a 5 second timer is started to retry.

External Services Need to Handle Many Simultaneous Clients

An external service needs to allow at least one connection from each datapath process. If the LineRate Scripting authenticates or identifies itself to the external service, the external service needs to allow multiple connections (may be called logins or sessions) from the same user at a time. Firewalls that sit between the LineRate and the external service should be configured so that they do not block multiple connection attempts from the LineRate to the external service.

My Compression Streams Never Complete

1. [Overview](#)
 2. [An Example of the Problem](#)
 3. [What's Going On?](#)
 4. [Diagnosing the Problem](#)
 5. [Handling the Problem](#)
-

Overview

Every zlib object inherits from the Node.js stream interface. As streams, zlib objects support the pipe facility as well as the pause and resume facilities. In addition, all zlib streams are Duplex, that is, each object is capable of reading and writing. This functionality allows for any number to be chained together one after the other.

The pipe facility lets you read data automatically from the previous stream, while automatically writing output to the downstream object. Without pipe, all data transfers would require bulky event registrations and data processing. To prevent slower points in the chain from consuming too many resources during a piped operation, streams support pausing and resuming behaviors. This means they provide back pressure on the chain, allowing the slower stream to pause upstream when writes come in too fast and to resume when it's caught up. Finally, a Duplex stream (supporting both [readability](#) and [writability](#)) can exist anywhere in the chain and is not contained solely at the head (Readable) or the end (Writable).

Take care to prevent complications from overusing these helpful shortcuts. A series of piped streams that creates a loop by accidentally reusing the same stream twice in the same chain can cause problems. Two common symptoms are streams that never complete and streams that produce errors or junk data.

A stream loop can stall and never complete. If, and probably when, the duplicated stream pauses the upstream object, it will trigger a cascade of pauses up the chain. This will eventually pause the duplicated stream itself (because of the loop in the chain). A paused stream cannot unpause itself. At this point, the entire operation is stalled permanently.

A stream can produce errors or junk data when it receives concurrent data from two different sources or data serialized from different sources without an intermediate reset. As stateful streams, zlib objects may expect certain inputs to satisfy the underlying algorithms. For example, pushing individual chunks of compressed data read from two network sockets into the same stream will cause a corrupted stream error. In addition, the decompression windows may not match, decompress or gzip headers will appear out of order, or required compression dictionaries may not exist, etc. An individual zlib stream must only receive data from one source at a time in between calls to `reset()`.

An Example of the Problem

The following example is highly contrived to elucidate the errors. It is foreseeable in the real world (through a chain of function callbacks) where this condition may occur outside of a straight serial chain. Callbacks can pass data to stream objects from various sources that can then confuse the compression/decompression state, or worse, introduce a cycle that stalls.

This script will push a large initial buffer into a cyclical chain, causing heavy back pressure and frequent reproduction of a permanently stalled operation or invalid stream states.

```
var vsm = require('lrs/virtualServerModule');
var zlib = require('zlib');

// This function chains multiple streams together
// erroneously duplicating one to cause a stall,
// or an error.
function response(servReq, servResp, next) {
  servReq.on('response', function(cliResp) {
    cliResp.bindHeaders(servResp);

    var gzip1 = zlib.Gzip();
    var gzip2 = zlib.Gzip();
    var gunzip = zlib.Gzip();
    cliResp
      .pipe(gzip1)
      .pipe(gunzip)
      .pipe(gzip2)
      .pipe(gunzip)
      .pipe(servResp);

    next();
  });
};

vsm.on('exist', 'vs1', function(vs) {
  vs.on('request', respond);
});
```

If you install this script and run it, you'll notice it will error with an "uncaught exception" with an error message or deadlock and fail to ever complete.

What's Going On?

With a very large file (one that might need compression on the wire), the first write from `cliResp` can be large, perhaps 1MB. A `zlib` stream (in this case `gzip1`) will process this data in 16 KB chunks, by default. Therefore, it will write to its downstream neighbor with buffers of 16 KB or less (possibly less for two reasons: the `libz` algorithm does not guarantee all data gets consumed in one pass, and the compression will decrease the data size). As `gzip1` consumes its buffer and writes, the downstream objects may apply back pressure and pause the upstream objects.

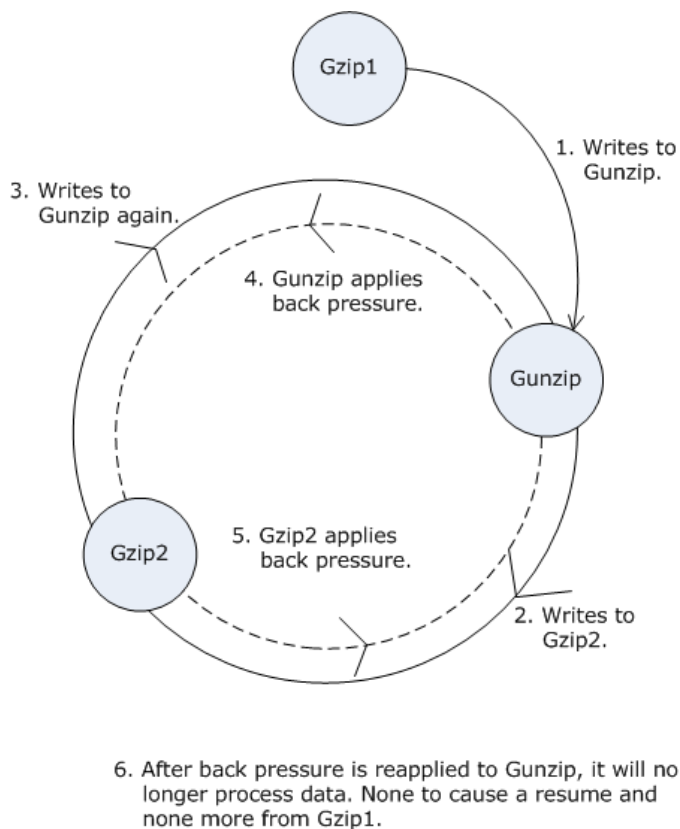
Two `zlib` behaviors are of note here:

- A `zlib` stream will consume its entire input buffer before pausing

- A zlib stream does not guarantee it will entirely consume that input buffer before writing/emitting data downstream.

Each downstream object may have multiple writes to it before upstream objects will pause, or the downstream object can consume what it currently has buffered to process.

In our example, when `gzip2` eventually requires itself to pause, it sends its pause message to `gunzip`. At this moment, the error may occur, and the `gunzip` can be concurrently paused downstream and written to from the upstream. This can put the object in an asymmetric state where it needs to process data but will short-circuit itself and never process the data. With `gunzip` being the final downstream object, it cannot drain and subsequently resume the `gzip2` stream. Now the entire chain is stalled.



Previous to the stall occurring, the `gunzip` stream can error and produce junk data. Notice that this chain would cause `gunzip` to receive concurrent writes from two different sources. Intermixing the two compression streams will conflate them in relation to `gunzip` and appear as one source of corrupted data.

Diagnosing the Problem

To determine if this condition is causing unexpected behavior in your script, look at the following:

- How long zlib streams references are kept.
- Are global references retained and the same object used multiple times?
- Are there any places in the script that explicitly call `reset()` on the object? If not double check each streams scope.
- Does the script pipe streams together through passed in data and callbacks? Try removing `pipe()` calls and replace them with manual data streaming via `on('data', cb)` and `on('end', cb)` calls. This will remove pauses, leaving the streams to run, but increase the likelihood that data errors occur with corruption faults. For example, this first statement can be written thusly:

```
gzip.pipe(gunzip);
```

Alternatively,

```
gzip.on('data', function(chunk) {
  gunzip.write(chunk);
})
.on('end', function() {
  gunzip.end();
});
gzip.end(buffer);
```

Handling the Problem

A zlib stream should best be considered a transitory utility to be created when necessary, used, and discarded. Using top-level stream objects from a pool can lead to data corruption issues and errors in scripts. Also, if you are using a large number of zlib pipes, the design of the script is in question. Oftentimes, input should be compressed on one side and decompressed on the other or vice versa.

Consider the following before using zlib streams:

- Question why zlib streams are in use where the input and output is both compressed or decompressed.
- Be certain not to chain multiple compression streams serially. Recompressing an already compressed stream can even allow an increase to the size of the ensuing data and should never be done.
- Make certain that your pipes are necessary and aren't filling a superfluous need.

To ease design, zlib streams should also be considered as associative objects and not independent from the source of their data. For example, if an application protocol is known to use compressed data, LineRate (as with SSL termination) can take the computational load of decompressing those transmissions before sending them to the application servers. In a scenario like this, connections and zlib streams should be logically associated together to ensure compression state integrity.

In short, zlib streams are inherently tied to their data source and tying their lifecycle to the data will prohibit the errors mentioned here.

My Request Handler Is Only Called Sometimes

1. [Overview](#)
 2. [Diagnosing the Problem](#)
 3. [Handling The Problem](#)
-

Overview

Each process registers request listeners independently. Virtual servers and forward proxies are configured independent of the number of processes on the system. The LineRate configuration system distributes information about all of the configured virtual servers and forward proxies to the processes. However, the scripts operate independently. The following example will only catch requests in process IDs that are even:

```
var vsm = require('lrs/virtualServerModule');

// BAD: vs1 may be created before the script
vsm.on('create', function (vs) {
  if (vs.id !== 'vs1') return;
  vs.on('request', function(req, res, next) {
    req.addHeader('X-Went-Through', 'RequestHandler');
    next();
  });
});
```

If you configure vs1 before you set the script source or turn it admin-online, the command **show script exampleScript** shows that no processes are listening for the request:

```
host-vm-1(config)# show script foo
Admin Status:      online
Restart Mode:      auto
Created At:        Wed Sep  4 15:19:10 2013 UTC
Attached Entities:
  Proxy Request Listeners
    <none>
Status:            UP
Last Runtime Error:
  <none>
```

If you deconfigure and reconfigure the virtual server after the script is configured, then it will listen:

```
host-vm-1(config)# no virtual-server vs1
host-vm-1(config)# virtual-server vs1
host-vm-1(config-vserver:vs1)# service http
host-vm-1(config-vserver-http:vs1)# attach virtual-ip vip1 default
host-vm-1(config-vserver:vs1)# show script foo
Admin Status:      online
Restart Mode:      auto
Created At:        Wed Sep  4 15:19:10 2013 UTC
Attached Entities:
```

```
Proxy Request Listeners
Name          Count
vs1           2 / 2
Status:       UP
Last Runtime Error:
<none>
```

These inconsistent results happen because of the order of events. The `vsm.on('create', ...)` event only happens when a virtual server is newly created.

Diagnosing the Problem

Examine calls to `vsm.on('create', ...)`, `vsm.find()`, `fpm.on('create', ...)`, `fpm.find()` where `vsm = require('lrs/virtualServerModule')` and `fpm = require('lrs/forwardProxyModule')`.

Consider what would happen if the virtual servers, forward proxies, and scripts were configured in unexpected orders.

Handling The Problem

Almost all uses of `vsm.on('create', ...)`, `vsm.find()`, `fpm.on('create', ...)`, `fpm.find()` where `vsm = require('lrs/virtualServerModule')` are better handled using the 'exist' event. See [Use 'exists' Event for Configuration Objects](#) for more details. Here is the example reworked to use exists:

```
var vsm = require('lrs/virtualServerModule');

// GOOD: exist event is tolerant of config in any order
vsm.on('exist', 'vs1', function (vs) {
  vs.on('request', function(req, res, next) {
    req.addHeader('X-Went-Through', 'RequestHandler');
    next();
  });
});
```

My Script Can Only Forward New Requests Sometimes

1. [Overview](#)
 2. [Diagnosing The Problem](#)
 3. [Handling The Problem](#)
-

Overview

Code at the outer scope, or called directly from the outer scope, is only called when the script is first configured. The following example tries to forward requests to a different virtual server:

```
var vsm = require('lrs/virtualServerModule');

// BAD: vs2 may not be configured before the script.
var vs2 = vsm.find('vs2');

vsm.on('exist', 'vs1', function (vs) {
  vs.on('request', function(req, res, next) {
    vs2.newRequest(req, res, next);
  });
});
```

This script will only work if vs2 is configured before the script is configured, because the `vsm.find()` executes once as soon as the script is configured and admin-status online.

Diagnosing The Problem

Examine the calls to `vsm.on('create', ...)`, `vsm.find()`, `fpm.on('create', ...)`, `fpm.find()` where `vsm = require('lrs/virtualServerModule')` and `fpm = require('lrs/forwardProxyModule')`. Consider what would happen if the virtual servers, forward proxies, and scripts were configured in unexpected orders.

Examine the outermost scope and ensure that everything that is in the outermost scope, or directly called by the outermost scope, meets these conditions:

- Only needs to happen once.
- Will succeed at the instant the script is configured.

Handling The Problem

Almost all uses of `vsm.on('create', ...)`, `vsm.find()`, `fpm.on('create', ...)`, `fpm.find()` where `vsm = require('lrs/virtualServerModule')` are better handled using the 'exist' event. See [Use 'exists' Event for Configuration Objects](#) for more details. Here is the example reworked to use exists:

```
var vsm = require('lrs/virtualServerModule');

// GOOD: vs2 starts undefined but as soon as it is created, it is assigned.
var vs2;
vsm.on('exist', 'vs2', function (vs) {
  vs2 = vs;
});

vsm.on('exist', 'vs1', function (vs) {
  vs.on('request', function(req, res, next) {
    // GOOD: script is tolerant of vs2 not being configured.
    if (vs2) {
      vs2.newRequest(req, res, next);
    } else {
      next();
    }
  });
});
```

User Sessions Are Spuriously Deleted

1. [Overview](#)

1.1. [What's Going On?](#)

1.2. [Diagnosing the Problem](#)

1.3. [Handling the Problem](#)

Overview

This is a behavior that is closely related to the behavior described in [A Variable Is Unset That I Know I Set](#). In this behavior, the LineRate Scripting sets cookies in the responses that are returned to a client to establish a session. Future requests from the client include the cookie. But, a surprising behavior occurs when the LineRate Scripting sometimes fails to look up the session information associated with the cookie.

The method of using a cookie to establish a session is well described [elsewhere](#). Briefly, the session cookie is returned to the client in the first response. The client includes the cookie in all of the future requests it makes, so the LineRate Scripting can map from the cookie to the user. The LineRate Scripting can check some state associated with the user. In this example, a script could check if the user is logged in and return an "Access Denied" page if they are not.

```
var vsm = require('lrs/virtualServerModule')
, urlMod = require('url')
, cookie = require('cookie');

var cookies = [];
function setCookie(req, res, next) {
  // BAD: This method is INSECURE and not suitable for mutliples processes.
  var session = { loggedIn: true, at: Date(), id: cookies.length};
  cookies.push(session);
  var responseBody = '<html><head><title>Logged In</title></head>\n' +
    '  <body><h2>Logged in, session ID: ' + session.id +
    '</h2></body></html>';
  res.setHeader('Set-Cookie',
    cookie.serialize('session', session.id, { 'path': '/', 'httpOnly': true }));
  res.setHeader('Content-type', 'text/html');
  res.setHeader('Content-length', responseBody.length);
  res.end(responseBody);
  next();
}

function getSession(req) {
  var parsed = cookie.parse(req.headers['Cookie']);
  if (parsed['session']) {
    return cookies[parseInt(parsed['session'])];
  }
  // Could not find cookie.
  return undefined;
}
```

```

function rejectNotLoggedIn(req, res, next) {
  var responseBody = '<html><head><title>Forbidden: Not Logged In</title></head>\n' +
    '  <body><h2>Forbidden: Not Logged in</h2></body></html>';
  res.writeHead(403, {'Content-type': 'text/html',
    'Content-length': responseBody.length });
  res.end(responseBody);
  next();
}

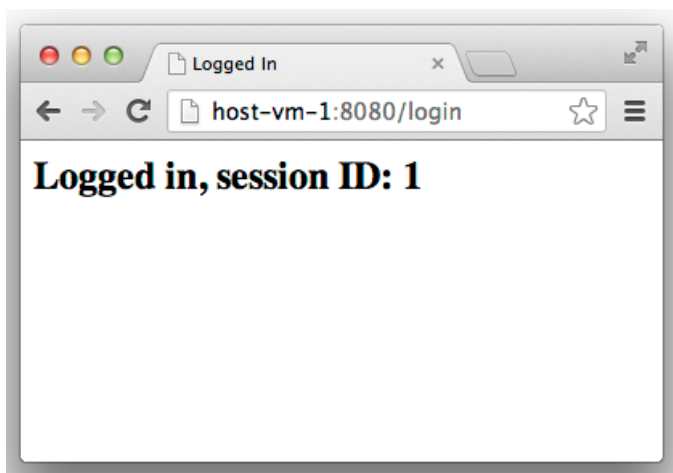
// A real proxy would pass this to the app server with next()
function respondLoggedIn(req, res, next) {
  var responseBody = '<html><head><title>Logged In</title></head>\n' +
    '  <body><h2>Logged in, ID: ' + req.session.id +
    ', since: ' + req.session.at + '</h2></body></html>';
  res.setHeader('Content-type', 'text/html');
  res.setHeader('Content-length', responseBody.length);
  res.end(responseBody);
  next();
}

function handleRequest(req, res, next) {
  var url = urlMod.parse(req.url);
  if (url.pathname === '/login') {
    setCookie(req, res, next);
    return;
  }
  req.session = getSession(req);
  if (!req.session) {
    rejectNotLoggedIn(req, res, next);
  } else {
    respondLoggedIn(req, res, next);
  }
  // Busy-loop for 200ms
  var start = Date.now();
  while (Date.now() < start + 200) {}
}

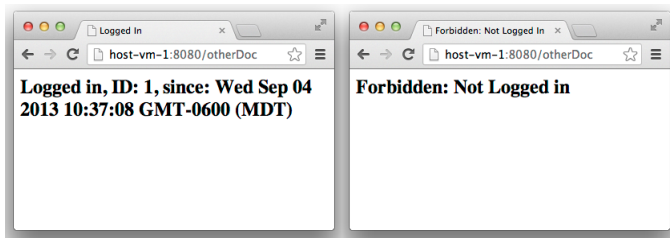
vsm.on('exist', 'vs1', function (vs) {
  vs.on('request', handleRequest);
});

```

When going to the /login page, your browser is given a new cookie:



Now, when going to any other path, your browser presents the cookie to the proxy. Sometimes, that path will load and return the "Logged in" page below left. Sometimes, the path will instead return the "Forbidden: Not Logged in" page below right, especially if the page is reloaded very quickly.



What's Going On?

The same underlying cause of the behavior from [A Variable Is Unset That I Know I Set](#) is at work here. One process sets the cookie on one request. As long as that process gets all the future requests from that user, the session information can be properly associated with that user.

As soon as another process happens to handle a request, it will be unable to find the session information associated with that cookie. The cookie itself is valid, as sent from the client to the proxy. But, the information associated with that cookie (stored in `var cookies`) only exists in the process that handled the login.

Diagnosing the Problem

The same principals for [Diagnosing The Problem](#) from [A Variable Is Unset That I Know I Set](#) previously should be applied. Try logging the PID associated with cookie creation and lookups. Here's a modification of the example script (look for the two "DIAGNOSIS" lines):

```
var vsm = require('lrs/virtualServerModule')
    , urlMod = require('url')
    , cookie = require('cookie');

var cookies = [];
function setCookie(req, res, next) {
  // BAD: This method is INSECURE and not suitable for mutliple processes.
  var session = { loggedIn: true, at: Date(), id: cookies.length };
  cookies.push(session);
  var responseBody = '<html><head><title>Logged In</title></head>\n' +
    '  <body><h2>Logged in, session ID: ' + session.id +
    '  </h2></body></html>';
  // DIAGNOSIS: Try logging when a new session is created, including the PID
  console.log('Creating session info for session ' + session.id +
    ' in PID ' + process.pid);
  res.setHeader('Set-Cookie',
    cookie.serialize('session', session.id, { 'path': '/', 'httpOnly': true }));
  res.setHeader('Content-type', 'text/html');
  res.setHeader('Content-length', responseBody.length);
  res.end(responseBody);
  next();
}
```

```

function getSession(req) {
  var parsed = cookie.parse(req.headers['Cookie']);
  if (parsed['session']) {
    var sessionId = parseInt(parsed['session']);
    var sessionInfo = cookies[sessionId];
    // DIAGNOSIS: Try logging when a lookup fails, including the PID.
    if (!sessionInfo) {
      console.log('Failed to lookup session info for session ' + sessionId +
        ' in PID ' + process.pid);
    }
    return sessionInfo;
  }
  // Could not find cookie.
  return undefined;
}

function rejectNotLoggedIn(req, res, next) {
  var responseBody = '<html><head><title>Forbidden: Not Logged In</title></head>\n' +
    ' <body><h2>Forbidden: Not Logged in</h2></body></html>';
  res.writeHead(403, {'Content-type': 'text/html',
    'Content-length': responseBody.length });
  res.end(responseBody);
  next();
}

// A real proxy would pass this to the app server with next()
function respondLoggedIn(req, res, next) {
  var responseBody = '<html><head><title>Logged In</title></head>\n' +
    ' <body><h2>Logged in, ID: ' + req.session.id +
    ', since: ' + req.session.at + '</h2></body></html>';
  res.setHeader('Content-type', 'text/html');
  res.setHeader('Content-length', responseBody.length);
  res.end(responseBody);
  next();
}

function handleRequest(req, res, next) {
  var url = urlMod.parse(req.url);
  if (url.pathname === '/login') {
    setCookie(req, res, next);
    return;
  }
  req.session = getSession(req);
  if (!req.session) {
    rejectNotLoggedIn(req, res, next);
  } else {
    respondLoggedIn(req, res, next);
  }
  // Busy-loop for 200ms
  var start = Date.now();
  while (Date.now() < start + 200) {}
}

vsm.on('exist', 'vs1', function (vs) {
  vs.on('request', handleRequest);
});

```

When the page is loaded several times until the "Forbidden" page arrives, the syslog has the culprit:

```

LROS: Creating session info for session 0 in PID 844
LROS: Failed to lookup session info for session 0 in PID 1378

```

The session information was created in PID 844. When PID 1378 attempted to look it up, it failed.

Handling the Problem

The section for [Handling the Problem](#) from the [A Variable Is Unset That I Know I Set](#) case provides two approaches to dealing with this issue. However, it will not be sufficient to [occasionally sync somewhere else](#). The session information should be [stored externally](#).

Managing Scripts Using the REST API

1. [Overview](#)
 2. [Uploading Scripts](#)
 3. [Enabling or Disabling Scripts](#)
-

Overview

In addition to using the CLI to create scripts, you can use the LineRate REST API. Using the REST API lets you automate the upload of scripts to multiple systems.

Uploading Scripts

You can manage the installation of scripts using the REST API to upload them as needed. This also makes it easier to automate uploads of scripts to multiple machines.

An example shell script uploading scripts via REST, [post-script.sh](#), is attached to this page. This script is meant run on UNIX/Linux systems under the bash shell. It uses curl to log in to the API and upload your script. Be sure to include the accompanying script, [jsonify.sh](#), in your \$PATH.

The core of this script is these few lines:

```
curl -ks -d "username=${USERNAME}&password=${PASSWORD}" -c ${COOKIEJAR} $LOGINPATH > /dev/null
echo -n "$SCRIPTNAME" | jsonify > $SCRIPTJSONFILE
curl -ks -b ${COOKIEJAR} -X POST -d "@${SCRIPTJSONFILE}" -H "Content-Type: application/json"
"$SCRIPTPATH" > /dev/null
cat $SCRIPTFILE | jsonify > $SCRIPTJSONFILE
curl -ksSf -b ${COOKIEJAR} -X PUT -d "@${SCRIPTJSONFILE}" -H "Content-Type: application/json"
"$SCRIPTPATH/$SOURCEPATH"
```

Below is the usage:

```
$ post-script proxyhost scriptname script.js
{"httpResponseCode":200,"requestPath":"/config/script/scriptname/
sourceInline","recurse":false}
```

This will upload the script script.js and name it "scriptname" in the config.

Enabling or Disabling Scripts

To enable and disable scripts, you can use the attached enable-script and disable-script programs, both of which use [setvalue.sh](#), which is used to set REST API values. Syntax is:

```
$ enable-script proxyhost scriptname
curl -k -c cookie.jar -d username=admin&password=changeme https://proxyhost:8443/
login
curl -k -X PUT -b cookie.jar -d @/tmp/post-script-scriptfile-QcI26l
https://proxyhost:8443/lrs/api/v1.0/config/script/scriptname/adminStatus -H Content-
Type: application/json
{"httpResponseCode":200,"requestPath":"/config/script/scriptname/
adminStatus","recurse":false}
```