Populating Tables With CSV Data Via Sideband Connections

George Watkins, 2012-02-02

Introduction

Datagroups and tables are the two primary methods we have in iRules for organizing key and value pairs. Both can be reused for subsequent connections. Datagroups have the advantage of being directly editable from the BIG-IP user interface, however they cannot be modified from within an iRule. This would open a potential security hole by allowing BIG-IP filesystem access from an iRule. Tables on the other hand must be populated from within an iRule, which allows for tracking user state, session data, etc. Table cannot be modified from any user interface though. With the advent of sideband connections, we can now make a connection to a separate service retrieve structured data and populate a table.

Tables were introduced in BIG-IP version 10 as a means to store key and value pairs across connections and sessions. Because tables are stored in memory, they are ultra-low latency and can handle large volumes of transactions. There are many cases when we want to functionality of a datagroup without requiring administrative access to the BIG-IP. With the introduction of sideband connections in version 11, we now have the ability to retrieve data from other sources. We can then parse that data and populate the table.

For this example, we've chosen the CSV (comma-separated value) file format to store the tabular data. CSV files are one of the oldest file structures for formatting tabular data. The standard is loosely described in RFC 4180, but there have been several interpretations of CSV structure in the past. We will be using the most common format whereby there is no header line, each field is separated by a comma, and each line terminated with carriage return and line feed (CR+LF).

Session Continuity, Table Locking, and Graceful Failures

The simplest implementation of this functionality would be to periodically delete all table data, retrieve new data, parse, and insert. This method presents a number of challenges for any virtual that is handling connections at a faster rate than the data retrieval can complete and even then this would be a poor implementation. As a remedy, we conduct the process differently, by only deleting old records after all new records have been added. This insures that there are no interruptions in the user experience.

Locking is another consideration that must be taken for high volume implementations. In order to prevent table corruption, we will lock the table to prevent two instances from modifying the same table simultaneously. We handle the locking mechanism using an entry in another table with a timeout and lifetime equal to the table's cache timeout.

Because we are modifying a potentially mission critical table, we want to handle any interruption in updates gracefully. Therefore, no existing data will be removed until we receive a valid set of new data. If a failure occurs, the next update attempt will not commence until the next scheduled refresh.

Caching

There is a small, but measurable amount of latency in triggering retrieval of new records. This latency was measured as less than one second for a sideband connection from an LTM virtual edition to a web server on the same LAN. In order to avoid each connection from performing this operation, we added a caching mechanism so updates are only initiated periodically. The length of time you'll want to cache records is dependent on your application. Some content such as a sitemap or a robot.txt file could be updated once daily. Other examples such as redirects or ACLs may need to be refreshed more regularly. This option is tunable via a static variable in RULE_INIT.

Putting It All Together

When we condense all of those provisions into a single process, we wind up with something that looks similar to the diagram below. This process outlines the procedure for importing CSV data. Each additional request during the cache period will bypass the import process and retrieve data directly from the table.

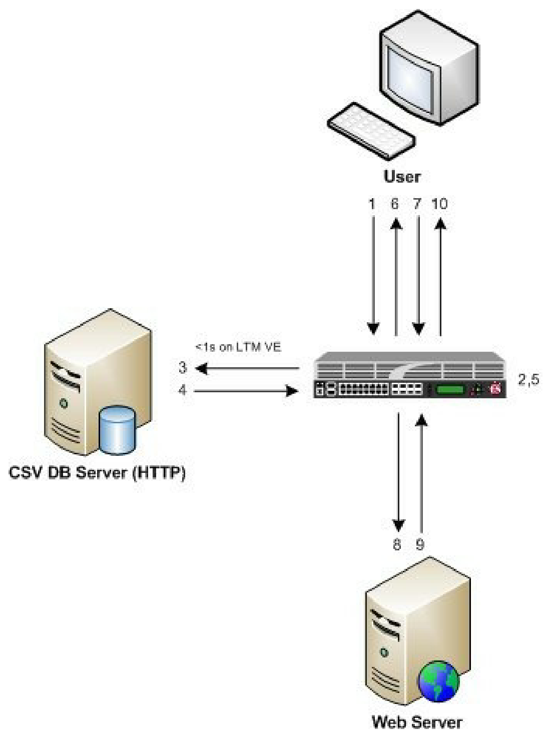

Table Import Process

1. User makes an HTTP request to virtual

2. iRule identifies expired table data and initiates refresh

3. LTM opens sideband HTTP connection to CSV DB server

4. DB server returns CSV data in response

5. iRule on LTM virtual will parse and insert the new data before removing any stale records; user's original request will proceed

6. User requested a redirect URL, so the LTM returns redirect with new URL

7. User initiates request for new URL

8. LTM passes request to origin server

9. Origin server responds to LTM with content

10. LTM responds with content to the user

The CSV Tabular Data Importer iRule

```
when RULE_INIT {
    # HTTP server holding the CSV-formatted database
    set static::db_host mydbhost.testnet.local

    # HTTP path for the CSV-formatted database
    set static::db_path "/redirects.csv"

    # CSV database line delimiter: CR = \r, LF = \n, CR+LF = \r\n
    set static::db_line_delimiter "\n"

    # DNS server if using DNS resolution (optional)
    set static::dns_server 10.0.0.1

    # Timeout for database cached in a table
    set static::db_cache_timeout 3600
}

when CLIENT_ACCEPTED {
    # table to cacha CSV-formatted database
    set db_cache_table "db_cache_[virtual]"

    # table to track when to refresh the database's contents
    set db_cache_state_table "db_cache_timeout_[virtual]"

    set last_refresh [table lookup -subtable $db_cache_state_table last_refresh]

    if { $last_refresh eq "" } { set last_refresh 0 }

    if { [expr [clock seconds]-$last_refresh] > $static::db_cache_timeout } {
        set db_ip [lindex [RESOLV::lookup @$static::dns_server -a $static::db_host] 0]

        if { $db_ip ne "" } {
            if { [table lookup -subtable $db_cache_state_table lock] != 1 } {
                # lock table modifications so that multiple instances don't attempt to update the table
                table set -subtable $db_cache_state_table lock 1 $static::db_cache_timeout $static::db_cache_timeout

                log local0. "Locking table"

                # establish connection to server
                set conn [connect -timeout 1000 -idle 30 $db_ip:80]

                # build request to send to HTTP server hosting DB
                set request "GET $static::db_path HTTP/1.1\r\nHost: $static::db_host\r\n\r\n"

                # send request to server
                send -timeout 1000 -status send_status $conn $request

                # receive response and place in variable
                set db_contents [getfield [recv -timeout 1000 -status recv_info $conn] "\r\n\r\n" 2]

                if { $db_contents ne "" } {
                    # update last refresh time in timeout table
                    table set -subtable $db_cache_state_table last_refresh [clock seconds] indef indef

                    # grab a list of old keys so we can remove them from cache if not in new DB copy
                    set old_keys [table keys -subtable $db_cache_table]

                    foreach field [split [string map [list $static::db_line_delimiter \uffff] $db_contents] \uffff] {
                        if { ($field contains ",") && !($field starts_with "#") } {
                            set sep_offset [string first "," $field]

                            set key [string range $field 0 [expr $sep_offset - 1]]
                            set value [string range $field [expr $sep_offset + 1] end]
                            lappend new_keys $key

                            # add key/value pairs to DB cache table
                            table set -subtable $db_cache_table $key $value indef indef

                            if { [lsearch $old_keys $key] >= 0 } {
                                log local0. "Updating \"$key\" = \"$value\" in DB cache table"
                            } else {
                                log local0. "Adding \"$key\" = \"$value\" to DB cache table"
                            }
                        }
                    }

                    foreach old_key $old_keys {
                        if { [lsearch $new_keys $old_key] < 0 } {
                            # remove any keys which don't exist in new DB copy
                            table delete -subtable $db_cache_table $old_key

                            log local0. "Deleting \"$old_key\" from DB cache table, key doesn't exist in new DB copy"
                        }
                    }
                }

                close $conn

                table delete -subtable $db_cache_state_table lock

                log local0. "Unlocking table"
            }
        } else {
            log local0. "Could not get valid IP for the DB server. Check the hostname and nameserver settings."
        }
    }
}

when HTTP_REQUEST {
    set redirect_path [table lookup -subtable $db_cache_table [string tolower [HTTP::path]]]

    if { $redirect_path ne "" } {
        HTTP::redirect http://[HTTP::host]$redirect_path
    }
}
```

Conclusion

The ability to populate tables with data via a sideband connection opens up many possibilities for handling application logic on the LTM. The biggest advantage is having an off-box method for adding data to iRules. We mentioned a few of the use cases such as redirects, sitemaps, and ACLs, but the list is really limitless. In addition to being able to manually edit the CSV files and import data to the LTM, there is now the capability to ingest dynamically generated data of any format. While the ability to perform redirects with an iRule may seem mundane, the underlying code opens up endless possibilities all of which translate to lower management effort. Until next time, happy coding!

Code and References

CSV Tabular Data Sideband Importer iRule - Documentation and code for the iRule used in this Tech Tip

RFC 4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files