|  |  |
| --- | --- |
| Current version | 1.0 DRAFT |

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Description of change |
| 0.9 | 6 October 2015 | Sriram Nambakam | Initial draft |
| 0.9 | 6 October 2015 | Kumar Kaushik | Adding use-case for LDAP |
| 0.9 | 29 October 2015 | Sriram Nambakam | Cleanup draft |
| 1.0 | 10 May 2017 | Kumar Kaushik | First Release |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Reviewers and Signoff History**

|  |  |  |  |
| --- | --- | --- | --- |
| Reviewer | Role | Comments | Date |
|  |  |  |  |

1. Summary 4

1.1. Overview 4

1.2. Purpose 4

1.2.1. Business Driver(s) 4

1.2.2. Technical Driver(s) 4

1.3. Goal 5

2. Problem Description 6

2.1. Problem 6

2.2. Current Product Limitations 6

2.3. Use Cases 7

2.3.1. LDAP over REST for VMware Directory 7

2.3.2. REST interface for VMware DNS 7

3. General Architecture. 8

3.1. Transport 8

3.2. Protocol heads 8

3.3. Handlers 9

3.4. Utilities 10

3.5. Sample Client 10

3.6. Sample Server 10

4. Product Requirements 11

4.1. Compatibility 11

4.2. Software Lifecycle 11

4.2.1. Installation 11

4.2.2. Deployment 11

4.2.2.1. Third party component deployment 11

4.2.3. Upgrade 11

4.2.3.1. Legacy 11

4.2.3.2. Future 11

4.2.4. Backup/Restore 11

4.3. Performance and Scalability 11

4.3.1. Large Scale Performance Considerations 11

4.3.2. Performance and Scalability Targets 12

4.4. User interfaces 12

5. Other Considerations 13

5.1. Licensing 13

5.2. Disk footprint 13

5.3. Memory footprint 13

6. References 14

7. Appendix A – Parsing HTTP(S) Headers 15

7.1. HTTP request message format 15

7.1.1. Request Line 15

7.1.2. Status/Response line 15

7.1.3. General Headers 16

7.1.4. Request Headers 16

7.1.5. Response Headers 17

7.1.6. Entity Headers 18

8. Appendix B – Utility APIs 19

8.1. Memory Management APIs 19

# Summary

## Overview

This document provides the specification for a minimal Representational State Transfer (ReST) engine that can be embedded in services that are implemented in ANSI C or C++.

## Purpose

### Business Driver(s)

User interfaces and other services require a HTTP(S) based REST interface for easy integration with our various existing services that are implemented in ANSI C or C++.

A ReST interface is preferred to easily negotiate proxies, firewalls and also provide easier integration with cloud enabled services.

### Technical Driver(s)

The primary technical drivers are the following.

* Size
* Performance
* Extensibility

## Goal

The primary deliverable is a shared library for the ReST engine. It will be supported on the following platforms.

* Linux
* Windows

The ReST Engine will support the following protocols.

* HTTP
* HTTPS

The following payload format will be supported.

* Any text based payload is supported.

The following components will be provided to support the ReST engine.

* A multi-threaded server that consumes the services of the ReST engine

# Problem Description

## Problem

VMware needs a library that implements a ReST server and can be easily embedded into various services implemented in ANSI C and/or C++.

The ReST engine must have the following attributes.

1. Small footprint.
2. Support HTTP as well as HTTPS over TCP
3. HTTP and HTTPS ports may be changed by consumer
4. Multi-threaded
5. Highly scalable (handle at least 10K client connections simultaneously)
6. Must be implemented using ANSI C to maximize portability and embedding.
7. The ReST Engine will provide utility APIs to help handle Authentication and Authorization using OAUTH and GSSAPI.

## Current Product Limitations

1. Only HTTP-1.1 will be supported
2. OpenSSL-1.x will be used to handle certificates
3. Consumers are expected to handle Authentication and Authorization using Kerberos/OAUTH etc.

## Use Cases

The typical use case for the ReST Engine is as follows.

1. At compile time, link with the ReST Engine shared library.
2. At runtime, call the initialization routine in the ReST Engine with configuration options. This will include options such as the ports being used for HTTP(S).
3. Register an endpoint with handlers with the ReST Engine.
4. Handle ReST calls within the registered handlers.
5. Unregister endpoints at any time, after which the handlers will not be called.
6. Call the uninitialized APIs in the ReST Engine before unloading the library (For instance, when the consumer process is shutting down).

### LDAP over REST for VMware Directory

The VMware Directory Service will serve LDAP requests over ReST. In order to achieve this, the directory service will embed the ReST engine.

### REST interface for VMware DNS

The contents of the VMware Domain Name Service may be managed through a ReST interface.

# General Architecture.

The ReST Engine is a library that is composed of the following modules.

1. Transport
2. Protocol heads for HTTP and HTTPS.
3. Handlers
4. Utilities

## Transport

The Transport module is responsible for the following functionality.

1. Bind to one or more network interfaces
2. Listen on specified ports for TCP based client connections
3. Hand off active connections to the consumer (for instance the protocol head)
4. Manage client connections at high scale (at least 10K simultaneous client connections)

In order to support high scale at high performance, the transport module will be implemented using the following implementation for managing network connections.

|  |  |
| --- | --- |
| **Platform** | **Implementation** |
| Linux | epoll |
| Windows | I/O Completion Ports |

## Protocol heads

The Protocol head will support the following protocols according to RFC 2616.

1. HTTP
2. HTTPS

The protocol head will handle the following tasks.

1. Handle certificate validation in the context of HTTPS
2. Parse and validate the ReST request
3. Create the ReST response as per wire format
4. Write back the ReST response using Transport Module APIs
5. Optionally support cookies
6. Call the appropriate registered handlers for each ReST request

The protocol head will not handle the following tasks.

* Parse or validate the payload

## Handlers

The consumer must register endpoints with the ReST Engine in order to process client requests. Each registration will consist of the following information.

1. Protocol { HTTP, HTTPS }
2. A URL fragment expression for the endpoint
3. A function table of callbacks for the following methods.
   1. HEAD
   2. GET
   3. POST
   4. PUT
   5. DELETE
   6. TRACE
   7. CONNECT
4. Parameters

A HTTPS registration is also required to provide a callback for Authentication.

The protocol head is expected to have parsed the headers in the HTTP Request before invoking the appropriate callback. The HTTP Request object cannot be modified by the handler.

Pointer to response object will be provided as argument in handler. User is not suppose to modify it at all.

The response object should be allocated using the utility APIs from the ReST Engine library provided for this purpose.

The request and response objects are opaque from the perspective of the handler callback functions.

## Utilities

The ReST Engine library will also provide utility APIs that fall into these categories.

1. HTTP Request Query APIs
2. HTTP Response Management APIs
3. Memory Allocation APIs
4. String management APIs

## Sample Server

A sample server is included in the implementation only to aid development.

The sample server is implemented using ANSI C.

The sample server is not included in the runtime deployment package of the ReST Engine. It will be included in the development package instead.

# Product Requirements

## Compatibility

The ReST Engine will support HTTP-1.1.

## Software Lifecycle

### Installation

The ReST Engine will be distributed in a separate RPM on Linux, as a Merge Module and MSI on Windows.

### Deployment

#### Third party component deployment

Third party components may bundle the ReST engine and deploy it as part of their package. It is also possible to download it through standard RPM package managers such as YUM, TDNF etc.

### Upgrade

#### Legacy

This is the first version of the ReST Engine. Therefore, the prospect of handling legacy versions does not have to be addressed yet.

#### Future

In the future, the ReST Engine is expected to support newer versions of HTTP, support different payloads, support more forms of authentication.

Future versions will be backwardly compatible.

### Backup/Restore

The ReST Engine does not save any configuration, state or data. Therefore, we do not foresee any significant handling for backup and restore.

## Performance and Scalability

### Large Scale Performance Considerations

The ReST Engine’s protocol head is expected to handle at least 10K client connections simultaneously. It is expected to use modern socket management implementations to support high scalability.

The ReST Engine is multi-threaded where the consumer can set the number of threads that must be a multiple of the number of cores on the system.

When handling large payloads, the ReST Engine will implement Zero-Copy support from the kernel (when available) to reduce buffer allocations.

### Performance and Scalability Targets

TBD

## User interfaces

The ReST Engine does not provide any User Interfaces.

# Other Considerations

## Licensing

The code will be distributed using the Apache 2.0 license

## Disk footprint

TBD

## Memory footprint

TBD

# References

1. HTTP-1.1 RFC 2616 (<http://www.w3.org/Protocols/rfc2616/rfc2616.txt>)
2. JSON RFC 4627 (<https://www.ietf.org/rfc/rfc4627.txt>)
3. JSON Data Interchange Format (<http://www.rfc-editor.org/rfc/rfc7159.txt>)
4. SPNEGO-based Kerberos and NTLM HTTP Authentication (<https://tools.ietf.org/html/rfc4559>)
5. Kerberos V5 (GSSAPI) Simple Authentication and Security Layer (SASL) Mechanism (<https://tools.ietf.org/html/rfc4752>)

# Appendix A – Parsing HTTP(S) Headers

## HTTP request message format

Request/Response <CR LF>

One or more general headers ( ‘:’ separated attribute-value pair) <CR LF>

One or more request/response headers ( ‘:’ separated attribute-value pair) <CR LF>

One or more entity headers ( ‘:’ separated attribute-value pair) < CR LF>

<CR LF>

Message-body

### Request Line

<Method> SP <Request-URI> SP <HTTP-Version> CRLF

* Methods

This indicated the operation to be performed on resource identified by Request URI.  
Following request tokens are supported by our rest engine.

GET  
POST  
DELETE

* Request-URI  
  This identifies the resource upon which request has to be applied.
* HTTP-Version  
  Only HTTP/1.1 is supported by rest engine.

### Status/Response line

Response Line = <HTTP-Version> SP <Status-Code> SP <Reason-Phrase> CRLF

* HTTP-Version  
  Only HTTP/1.1 is supported by rest engine.
* Status Code   
  - 1xx: Informational - Request received, continuing process.   
  - 2xx: Success - The action was successfully received, understood, and accepted.   
  - 3xx: Redirection - Further action must be taken in order to complete the request.   
  - 4xx: Client Error - The request contains bad syntax or cannot be fulfilled.  
  - 5xx: Server Error - The server failed to fulfill an apparently valid request.

* Reason-Phrase  
  Please refer RFC 2616.

### General Headers

These header fields are applicable for both request and response messages but does not apply on entity being transferred.

Following general headers are supported by rest engine.

* Cache control  
  TBD
* Connection  
  Connection status after completion of response. This helps in identifying persistent versus non-persistent connection which are necessary for rest engine end use-case.
* Trailer  
  To be used with chunk transfer encoding helping recipient to know what headers field are expected in the trailer.
* Transfer encoding.  
  This header field indicates what transformation has been applied to the message for transfer between sender and receiver.

### Request Headers

These are additional header fields in request messages which allows clients to pass additional information about the request and client itself to the server.

Following request headers are supported by restengine.

* Accept.  
  Use to specify media type which are acceptable in response. For rest engine case this should be JSON. The syntax for this header field is little different from attribute-value pair.
* Accept –charset  
  This tell server which character set is supported at client side so that response can be understood.
* Accept-encoding  
  This tell content –coding that are acceptable in response.
* Accept-Language.  
  This restricts the set of natural language that are preferred as a response to request.
* Authorization

TBD

* From  
  Human readable email address of client using the REST services.
* Host  
  Specifies the internet host and port number of resource being requested. This is obtained from URI given by user.
* Referer  
  This tells server what URL lead the request to arrive at server.

### Response Headers

These are the header, which lets server to pass additional information in the response message, which cannot be provided in the response/status line.

Rest engine supports following response headers.

* Accept range.  
  Allows the server to indicate its acceptance of range request for resource.
* Location

This header field is used for redirecting the recipient to a location other tha n request URI for completion of request. This can be used with 201 (creation) or 3XX response code.

* Proxy-Authenticate.  
  TBD
* Retry-After  
  This can be used with 503 service unable response code to indicate how long service is expected to be unavailable to the requested client.
* Server.  
  The Server response-header field contains information about the software used by the origin server to handle the request. For case of rest engine this can be name of service used to query the request. (LDAP, VMAFD, VMCA etc.)

### Entity Headers

These are the headers fields associated with the entity body commonly known as payload. This can be used with both request and response messages both.

Rest engine supports following entity headers.

* Allow  
  This lists the set of methods supported by the resource identified. This must be used by 405 status message. Currently our rest engine support GET, POST and DELETE methods only.
* Content-Encoding  
  This specifies the additional content encoding that has been applied to entity body helping in the decoding of media type.
* Content-Language  
  Specifies the natural language of enclosed entity.
* Content-length  
  This indicates the size of entity body in decimal number of Octets.
* Content-Location  
  If location of resource is different from the one which specified in URL.
* Content-MD5  
  TBD
* Content-Range  
  This is used in case of partial entity body to tell where in full entity, this partial entity has to be applied.
* Content-Type  
  Specifies the media type which in case of rest engine would be JSON.

# Appendix B – Utility APIs

## Memory Management APIs

/\*

\* @brief Allocation of heap memory for rest engine.

\*

\* @param[in] size of memory to be allocated

\* @param[out] pointer to allocated memory

\* @return Returns 0 for success

\*/

uint32\_t

VmRESTAllocateMemory(

size\_t dwSize,

void\*\* ppMemory

);

/\*

\* @brief Free of head memory for rest engine.

\*

\* @param[in] pointer to allocated memory

\* @return Returns 0 for success

\*/

void

VmRESTFreeMemory(

void\* pMemory

);