VTS3 Communication Specification

Version 1.1

Contents

[1. Introduction 2](#_Toc370378559)

[2. Request and response 2](#_Toc370378560)

[2.1. Request syntax 2](#_Toc370378561)

[2.2. Response syntax 3](#_Toc370378562)

[2.3. Sample 4](#_Toc370378563)

[3. Technical details 4](#_Toc370378564)

[3.1. Commands details 4](#_Toc370378565)

[3.1.1. "handshake" command 4](#_Toc370378566)

[3.1.2. "add" command 5](#_Toc370378567)

[3.1.3. "get" command 6](#_Toc370378568)

[3.1.4. "retrieve" command 7](#_Toc370378569)

[3.1.5. "update" command 8](#_Toc370378570)

[3.1.6. "update\_ifequals" command 9](#_Toc370378571)

[3.1.7. "clear" command 9](#_Toc370378572)

[3.1.8. "inc" command 10](#_Toc370378573)

[3.1.9. "createColumn" command 10](#_Toc370378574)

[3.1.10. "clearColumn" command 11](#_Toc370378575)

[3.1.11. "getColumSize" command 11](#_Toc370378576)

[3.1.12. Index APIs 12](#_Toc370378577)

[3.1.13. Other APIs 12](#_Toc370378578)

[3.2. Debugging Tips 12](#_Toc370378579)

[3.2.1. Set log level 12](#_Toc370378580)

[3.2.2. Using API test page 13](#_Toc370378581)

[3.2.3. Running VTS with the command line. 14](#_Toc370378582)

[4. Sample VTS client code 15](#_Toc370378583)

# Introduction

Virtual Table Server (VTS) is a tool that shares data and parameters between LoadRunner vusers. As part of the new version 11.52 of LoadRunner we have created a new Virtual Table Server (VTS3). Its functionality is based on current VTS (Virtual Table Server) 2.1.

This new VTS uses web APIs for communication, which uses JSON and HTTP as transport. This document talks about communication specification, between VTS API and VTS server. Other developers can write their own client to the VTS server based on the spec, if they need to their client outside LoadRunner.

VTS use HTTP to do the communication, and below are some characteristics of the communication.

* Method: POST
* Content Type: application/x-www-form-urlencoded
* Server URL: <http://host:port>/api/
* Request format: JSON
* Response format: JSON

Please note that in URL, the relative path "/api" is used for accessing API functionality

For simplicity, all the data are put in HTTP message body, and is in JSON format. Below we will discuss the format of HTTP request and response.

# Request and response

## Request syntax

For communication protocol, the HTTP body of request message matches the following format:

request={  
 "cmd":<command name>,  
 "version":<version string>,  
 "data":<command argument>  
}

|  |  |
| --- | --- |
| cmd | Command name can be any of the following:  handshake | add | get | retrieve | update | clear | createColumn | clearColumn | getColumnSize | ensureIndex | dropIndex |
| version | Protocol version of the current message. 1.0 for the first version. |
| data | format is different depends on the cmd name |

Below is the BNF syntax:

"cmd":<command name>

<command name> := "handshake" | "add" | "get" | "retrieve" | "update" | "update\_ifequals" | "clear" | "createColumn" | "clearColumn" | "getColumnSize" | "ensureIndex" | "dropIndex"  
 <version string> := "1.0"  
 "data": "{" <command arguments> "}"

"<command arguments>" syntax varies for different command, and they are explained below:

|  |  |
| --- | --- |
| **Command Name** | **"data" Syntax** |
| "handshake" | <command arguments> := "version": "1.0" |
| "add" | <command arguments> :=  "new": <row object>,  "option": <option value>  <option value> := 0  *Notes: option value can only be 0 for "add" command* |
| "get" | <command arguments> := "index": <row index>  [, "columns": <row object>] |
| "retrieve" | <command argument> := ["columns": <row object>]  *Notes: if <command argument> is empty, will retrieve the whole row* |
| "update" | <command arguments> :=  "new": <row object>,  "option": <option value>  <option value> := 1  *Notes: option value can only be 1 for "update" command* |
| update\_ifequals | <command arguments> :=  "new": <row object>,  "option": <option value>  <option value> := 2  *Notes: option value can only be 2 for "*update\_ifequals*" command* |
| "clear" | <command arguments> := "index": <row index>  [, "columns": <row object>] |
| "createColumn" | <command arguments> := "columns": <row object> |
| "clearColumn" | <command arguments> := ["columns": <row object>]  *Notes: if <command argument> is empty, will clear the entire table* |
| "getColumnSize" | <command arguments> := "column": <column name> |
| "ensureIndex" | <command arguments> := "columns": "[" < column name > (","<column name>)\* "]" |
| "dropIndex" | <command arguments> := "columns": "[" < column name > (","<column name>)\* "]" |

Some definition used above:

<row object> := "{" <key value pair> (","<key value pair>)\* "}"  
 <key value pair> := <column name> : <column value>

For the detailed format of each command, please refers to section "**3. Commands Details**"

## Response syntax

The message response body matches the following format:

{  
 "data":<object>,  
 "status":{"error":<string>, "code":<number>}  
}

|  |  |
| --- | --- |
| data | "data" is also JSON object corresponding to the cmd in the request.  In case of error, data is set to **null**.  When row data is returned, the format is consistent with "<row object>" format mentioned in **2.1 Request Syntax** section |
| status.error | error messages |
| status.code | 0 for success, negative number for fail |

## Sample

Here is a sample request and response for getting the first row from the VTS table

**Request:**

request={

"cmd": "get",

"version": "1.0",

"data": {

"index": 1

}

}

**Response:**

{

"data": {

"ProductID": "1",

"ProductName": "Chai",

"SupplierID": "1",

"CategoryID": "1",

"QuantityPerUnit": "10 boxes x 20 bags",

"UnitPrice": "18",

"UnitsInStock": "39",

"UnitsOnOrder": "0",

"ReorderLevel": "10",

"Discontinued": "false"

},

"status": {

"error": null,

"code": 0

}

}

# Technical details

## Commands details

### "handshake" command

"handshake" is the process to tell server the client protocol version, and also get server protocol version. The "Data" field has the following format:

"data": { "version": "1.0" }

Version "1.0" communication is used in new VTS released with LoadRunner 11.52. "1.1" is the communication of VTS in 12.0 (to be released). The spec we discussed in this spec is 1.0, which is fully compatible with 1.1.

During handshake, the server side can determine whether to raise any error for incompatible client version. This is designed for future protocol upgrade. Currently, using 1.0 with 1.1 server is fully compatible.

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "handshake",  "version": "1.0",  "data": {  "version": "1.0"  }  } | {  "data": {  "version": "1.0"  },  "status": {  "error": "OK",  "code": 0  }  } |

VTS data APIs have no server sessions, which means if you want to implement your own version of the following APIs with your language, your main job is to manage the client TCP connections, e.g. KEEP\_ALIVE flag.

* vtc\_connect
* vtc\_disconnect
* lrvtc\_connect
* lrvtc\_disconnect

### "add" command

Add the new object to database according to the option field.

"data" field has the following format:

"data": {

"new": {

"column1": "value1",

"column2": "value2",

"column3": "value3",

...

},

"option": 0

}

"option" has the following values:

* 0 : add as same row
* 1 : add as stack
* 2 : add as unique stack

The following is a command that is similar to vtc\_send\_row1.

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**add**",  "version": "1.0",  "data": {  "new": {  "UserName": "Jason",  "Password": "hello123",  "Email": "asdf@hp.com"  },  "option": 0  }  } | {  "data": null,  "status": {  "error": null,  "code": 0  }  } |

The following is a command that is similar to vtc\_send\_if\_unique. Note the option is 2 in request.

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**add**",  "version": "1.0",  "data": {  "new": {  "columnName": "hello"  },  "option": 2  }  } | {  "data": 1,  "status": {  "error": null,  "code": 0  }  }  Or  {  "data": 0,  "status": {  "error": null,  "code": 0  }  }  Depends on whether the data is unique in this column. "1" means the newly added value is unique and added successfully, otherwise it is non-unique. |

If a specific column has never been defined before (neither defined during data import nor sent through script API), it will be created on the server as an independent collection/table automatically.

The following APIs are using "add" command:

* vtc\_send\_message
* vtc\_send\_if\_unique
* vtc\_send\_row1
* lrvtc\_send\_message
* lrvtc\_send\_if\_unique
* lrvtc\_send\_row1

### "get" command

Gets the values for the specified columns and row index. When data.columns is null, [], or undefined, server will return the entire row.

If for a certain column, there is no value for the given row index, return null for data field.

The following is a command that is similar to **vtc\_query\_row**, which can get a whole row of data of the 1st row:

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**get**",  "version": "1.0",  "data": {  "index": 1  }  } | {  "data": {  "ProductID": "1",  "ProductName": "Chai",  "SupplierID": "1",  "CategoryID": "1",  "QuantityPerUnit": "10 boxes x 20 bags",  "UnitPrice": "18",  "UnitsInStock": "39",  "UnitsOnOrder": "0",  "ReorderLevel": "10",  "Discontinued": "false"  },  "status": {  "error": null,  "code": 0  }  } |

The following is a command that is similar to **vtc\_query\_column**, which can get some specific columns value of the 1st row. It get only columns **ProductID**, **ProductName** and **UnitPrice** of the first row:

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**get**",  "version": "1.0",  "data": {  "columns": [  "ProductID",  "ProductName",  "UnitPrice"  ],  "index": 1  }  } | {  "data": {  "ProductID": "1",  "ProductName": "Chai",  "UnitPrice": "18"  },  "status": {  "error": null,  "code": 0  }  } |

The following VTS APIs are using "get" command:

* vtc\_query\_column
* vtc\_query\_row
* lrvtc\_query\_column
* lrvtc\_query\_row

### "retrieve" command

Remove the first row of specified columns and return the removed values. When data.columns is null, [], get entire row.

If for a certain column, there is no value, return null for it.

The following is a command that is similar to vtc\_retrieve\_row, which can return the entire 1st row, and remove it from database. Note that in request, it does not specify any column names, therefore the entire row are returned.

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**retrieve**",  "version": "1.0",  "data": {}  } | {  "data": {  "ProductID": "1",  "ProductName": "Chai",  "SupplierID": "1",  "CategoryID": "1",  "QuantityPerUnit": "10 boxes x 20 bags",  "UnitPrice": "18",  "UnitsInStock": "39",  "UnitsOnOrder": "0",  "ReorderLevel": "10",  "Discontinued": "false"  },  "status": {  "error": null,  "code": 0  }  } |

The following is a sample request/response that is similar to vtc\_retrieve\_message, which can get some column values of the 1st row. It specifies any column names as an array. As the result these two columns values are returned and removed from the database.

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "retrieve",  "version": "1.0",  "data": {  "columns": [  "ProductName",  "CategoryID"  ]  }  } | {  "data": {  "ProductName": "Chai",  "CategoryID": "1"  },  "status": {  "error": null,  "code": 0  }  } |

The following APIs are using "retrieve" command:

* vtc\_retrieve\_message
* vtc\_retrieve\_messages1
* vtc\_retrieve\_row
* lrvtc\_retrieve\_message
* lrvtc\_retrieve\_messages1
* lrvtc\_retrieve\_row

### "update" command

Updates data located in the specified columns and index.

If the row index is beyond row count for a specific column, then a new row of that index will be inserted. It will also insert rows between the index and the previous last row if they don’t exist.

The following sample update the second row of "ProductName" column to value "Tea".

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**update**",  "version": "1.0",  "data": {  "new": {  "ProductName": "Tea"  },  "index": 2  }  } | {  "data": null,  "status": {  "error": null,  "code": 0  }  } |

This command is used by the following VTS APIs:

* vtc\_update\_message
* vtc\_update\_row1
* lrvtc\_update\_message
* lrvtc\_update\_row1

### "update\_ifequals" command

Updates data located in the specified columns and index if the specified condition is satisfied.

If the row index is beyond row count for a specific column, then a new row of that index will be inserted.

The following sample will check the first row of column "columnName", if the value is "hello", will change the value to "world". In response, the data will be 1 if update is successful, otherwise, it will be 0.

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**update\_ifequals**",  "version": "1.0",  "data": {  "new": {  "columnName": "world"  },  "condition": {  "columnName": "hello"  },  "index": 1  }  } | {  "data": 1,  "status": {  "error": null,  "code": 0  }  } |

This command is used by the following VTS API:

* vtc\_update\_message\_ifequals

### "clear" command

Sets value to empty string for given row of specified columns

The following sample will clear the content of first row, but will not remove the cells of the first row.

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**clear**",  "version": "1.0",  "data": {  "index": 1  }  } | {  "data": null,  "status": {  "error": null,  "code": 0  }  } |

The following sample will clear the second row content the three columns:

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**clear**",  "version": "1.0",  "data": {  "columns": [  "ProductID",  "ProductName",  "SupplierID"  ],  "index": 2  }  } | {  "data": null,  "status": {  "error": null,  "code": 0  }  } |

This command is used by the following VTS APIs:

* vtc\_clear\_message
* vtc\_clear\_row

### "inc" command

Increments a counter at a specified column and index and returns the new number.

Client "add" value to a cell. Then when Client performs an "inc" command on this cell, this cell is queried out and parsed as an integer, then increment it and store the new value back. This will run in on server side as an atomic operation.

The following sample increments the first row of column "UnitPrice" by 10. The original value is 18. So in response, it returns the new value of 28.

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**inc**",  "version": "1.0",  "data": {  "value": 10,  "column": "UnitPrice",  "index": 1  }  } | {  "data": 28,  "status": {  "error": null,  "code": 0  }  } |

This command is used by the following VTS APIs:

* vtc\_increment
* lrvtc\_increment

### "createColumn" command

Creates columns defined in columns field but not add any values. Client doesn’t need to create a column before adding values to the column. But sometimes, some application may prefer to create table schema before using it, so they can use this command.

The following sample creates two new columns:

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "createColumn",  "version": "1.0",  "data": {  "columns": [  "Brand",  "manufacturer"  ]  }  } | {  "data": null,  "status": {  "error": null,  "code": 0  }  } |

This command is used by the following VTS APIs:

* vtc\_create\_column
* lrvtc\_create\_column

### "clearColumn" command

Clears the given columns entirely, but will still keep the columns definition.

The following sample clear two columns data.

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**clearColumn**",  "version": "1.0",  "data": {  "columns": [  "SupplierID",  "CategoryID"  ]  }  } | {  "data": null,  "status": {  "error": null,  "code": 0  }  } |

The following sample didn’t specify any column names, and will clear the entire table:

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**clearColumn**",  "version": "1.0",  "data": {}  } | {  "data": null,  "status": {  "error": null,  "code": 0  }  } |

This command is used by the following VTS APIs:

* vtc\_clear\_column
* lrvtc\_clear\_column

### "getColumSize" command

Gets row counts of the column field. For example:

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "**getColumnSize**",  "version": "1.0",  "data": {  "column": "ProductID"  }  } | {  "data": 77,  "status": {  "error": null,  "code": 0  }  } |

This command is used by the following VTS APIs:

* vtc\_column\_size
* lrvtc\_ column\_size

### Index APIs

vtc\_ensure\_index communication sample:

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "ensureIndex",  "version": "1.0",  "data": {  "columns": [  "ProductID"  ]  }  } | {  "data": null,  "status": {  "error": null,  "code": 0  }  } |

vtc\_drop\_index communication sample:

|  |  |
| --- | --- |
| **Request** | **Response** |
| request={  "cmd": "dropIndex",  "version": "1.0",  "data": {  "columns": [  "ProductID"  ]  }  } | {  "data": null,  "status": {  "error": null,  "code": 0  }  } |

### Other APIs

The following APIs doesn’t have a corresponding communication command because they are just client side operation:

* vtc\_free
* vtc\_free\_list
* vtc\_noop
* lrvtc\_noop

## Debugging Tips

If you have created a client for VTS, and want to debug it with a VTS server, this section contains some tips:

### Set log level

Logging may be configured in "logger" section of configure.json. See the following:

"logger":{

"level":"error",

"transports":{

"file":{

"type": "File",

"path": "%TEMP%/VTS"

},

"console":{

"type": "Console"

}

}

}

* "level" can be "info", "error", "warning".
* "type" can be "File", "Console"

The path may contain environment variable reference. Note, when running VTS Server as SYSTEM account, the %TEMP% is actual c:\windows\Temp

### Using API test page

You can access it with the following steps:

* Go to C:\Program Files (x86)\HP\VTS\web, open configure.json with editor, change "enableDiag" to true, and save the file.
* Restart VTS service by running **net stop "vts service"** and **net start "vts service"**.
* Browse to admin port: http://<vtsserver>:4000/data/diag, and click "apis", this page can be used as the test bed of API calls using JSON. E.g. the get\_message\* api can be tested like the following screen.

If you haven’t enabled API access, you may get the following message when try to run the command from the test page:

{"code":"ECONNREFUSED","errno":"ECONNREFUSED","syscall":"connect"}

Below is the test page to test API (the UI may look slightly different with different versions of VTS):



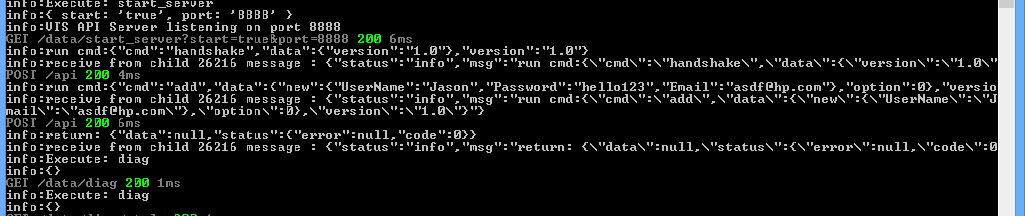
### Running VTS with the command line.

Another trick to detect any JSON communications of any VTS API calls:

1. On VTS installation folder, change the log level from "error" to "info"

                    "level":"error", /\* three levels: error, warning, info \*/

1. Run VTS from command line (please first stop the windows service to avoid port conflict). By navigating to the VTS installation folder (usually it is c:\Program Files (x86)\hp\vts\web\), and type "**node engine\main.js**" or "**node main.js**" (depends on your VTS version), you may need to change to the folder that contains node.exe and main.js file.
2. Then when you call an API from LR script, you will see the request and response printed on the console window, like the following screen:



It can shows you the request and response of the JSON call.

# Sample VTS client code

You can use the below sample JavaScript code to test VTS client server communication, with the following steps:

1. Save the below content to js file, e.g. vtsclient.js
2. Run "node vtsclient.js" to test it.

The sample is in JavaScript, but should be easier to convert to other languages

Note:

* You can download **node** engine from <http://nodejs.org/>, or find the file from VTS installation.
* You can change host and port defined at the beginning of the file.
* You can add more API testing code at the end of the file (more callApi function calls).

|  |
| --- |
| //please change VTS host and API port accordingly.  var host = 'localhost'  var port = 8888;  var http = require('http');  function createRequest(cmd, data) {  return { cmd: cmd, version: '1.0', data: data };  }  function sendRequest(request, callback) {  var req = http.request({ port: port, host: host, method: 'post', path: '/api', headers: { 'content-type': 'application/x-www-form-urlencoded' } });  req.end(JSON.stringify({'request=' + JSON.stringify(request) }));  req.on('response', function (response) {  response.on('data', function (data) {  callback(JSON.parse(data.toString()));  });  });  }  function callApi(cmd, data) {  var request;  sendRequest(request = createRequest(cmd, data), function (data) {  console.log('Request: \n' + JSON.stringify(request, null, 4));  console.log('Response: \n'+ JSON.stringify(data, null, 4));  console.log('\n\n');  });  }  //call VTS server with commands and parameters(data object)  //it will print in console  callApi('get', { index: 1 });  callApi('getColumnSize', { column: 'ProductID' }); |

After run, it will print in console window the request and response

