HTTP Plugin for MySQL

HTTP Plugin for MySQL

Abstract

The HTTP Plugin for MySQL adds HTTP(S) interfaces to MySQL. Clients can use the HTTP respectively HTTPS (SSL) protocol to query data stored in MySQL. The query language is SQL but other, simpler interfaces exist. All data is serialized as JSON.

This version of MySQL Server HTTP Plugin is a Labs release, which means it's at an early development stage. It contains several known bugs and limitation, and is meant primarily to give you a rough idea how this plugin will look some day.

Likewise, the user API is anything but finalized. Be aware it will change in many respects.

Document generated on: 2014-09-22 (revision: 293)

Table of Contents

Chapter 1 Overview

The HTTP Plugin for MySQL is a proof-of concept of a HTTP(S) interface for MySQL 5.7.

The plugin adds a new protocol to the list of protocols understood by the server. It adds the HTTP respectively HTTPS (SSL) protocol to the list of protocols that can be used to issue SQL commands. Clients can now connect to MySQL either using the MySQL Client Server protocol and programming language-dependent drivers, the MySQL Connectors, or using an arbitrary HTTP client.

Results for SQL commands are returned using the JSON format.

The server plugin is most useful in environments where protocols other than HTTP are blocked:

- JavaScript code run in a browser
- an application server behind a firewall and restricted to HTTP access
- a web services oriented environment

In such environments the plugin can be used instead of a self developed proxy which translates HTTP requests into MySQL requests. Compared to a user developed proxy, the plugin means less latency, lower complexity and the benefit of using a MySQL product. Please note, for very large deployments an architecture using a proxy not integrated into MySQL may be a better solution to clearly separate software layers and physical hardware used for the different layers.

The HTTP plugin implements multiple HTTP interfaces, for example:

- plain SQL access including meta data
- a CRUD (Create-Read-Update-Delete) interface to relational tables
- an interface for storing JSON documents in relational tables

Some of the interfaces follow Representational State Transfer (REST) ideas, some don't. See below for a description of the various interfaces.

The plugin maps all HTTP accesses to SQL statements internally. Using SQL greatly simplifies the development of the public HTTP interface. Please note, at this early stage of development performance is not a primary goal. For example, it is possible to develop a similar plugin that uses lower level APIs of the MySQL server to overcome SQL parsing and query planning overhead.

Chapter 2 Basic concepts

Table of Contents

2.1 How it works

MySQL supports a plugin API to create server components. There are several types of plugins such as storage engine plugins, full-text parser plugins, authentication plugins, daemon plugins and many more. Daemon plugins are not tailored to the creation of any specific server capabilities. A daemon plugin can be used for any code that should be run by the server.

The HTTP plugin for MySQL is a daemon plugin. Daemon plugins, like all other plugins, run within the process scope of the MySQL server.

To illustrate the working of the HTTP plugin, it can be considered to consist of two internal modules: a HTTP server module and a core module.

The HTTP server module is started when the plugin is loaded into MySQL. Upon start, it begins listening for HTTP requests, parses them and invokes callbacks of the core module to reply to them. The callbacks in the core module then decide whether to return error messages, deny access or execute SQL statements and return the results to the HTTP client through the HTTP server module. The HTTP server module implements different user APIs for different use cases. The internal use of SQL is exposed to a different degree. For example, the JSON document API aims to hide the use of SQL from the user altogether.

The HTTP server module is build using the PION HTTP library. PION is C++ library for developing HTTP services. The HTTP library is multi-threaded and makes use of asynchronous APIs.

Plugins can use assorted plugin services built-in to and provided by MySQL. The HTTP plugin is using a new SQL execution plugin service. The service allows plugins to execute arbitrary SQL statements as a certain MySQL user through a well-defined API. MySQL returns the results of the SQL statements executed as a binary stream to the plugin.

The HTTP plugin core module is using the service to answer HTTP requests. All HTTP requests are mapped to appropriate SQL statements. Then, the HTTP plugin core module serializes the results as JSON.

The new SQL execution plugin service is still under development and only available with the server in this Lab release. This is why you cannot use the HTTP Plugin for MySQL preview with any other MySQL server but the bundled one.

Following the pattern described it is possible to extend MySQL with other network protocols. For example, one could develop a daemon plugin that combines the lightweight and fast websocket protocol with the rich query capabilities of SQL.

2.2 Security compared with a web service

The HTTP plugin offers a similar level of security than a standard multi-tier web service build atop of MySQL.

When the HTTP plugin for MySQL receives a request, the request is authorized using HTTP specific methods. The only HTTP authentication mechanism supported by the HTTP plugin Lab release is HTTP basic authentication. HTTP basic authentication is a weak yet simple to use method.

HTTP(S)/SSL can be used encrypt the communication channel. Please note, the Lab release does not include SSL support for packaging reasons. SSL support is currently only available when using OpenSSL. This release does not include OpenSSL support.

After successful HTTP authorization against the web service, methods of the web service are invoked, which may further validate and sanitize the request. The same happens when using the HTTP plugin. The HTTP request is parsed, checked for validity and mapped to built-in methods.

In a second step methods of the web service respectively methods of the HTTP plugin may execute SQL statements. Those statements are executed on behalf of a certain MySQL user. The user must be authenticated and authorized against MySQL. Standard MySQL user management features can be used to grant the user only the permissions needed.

The web service and the HTTP plugin differ in the way they connect to the MySQL server. The web service has to establish an external communication channel to MySQL, which should be secured. For example, SSL can be used. The HTTP plugin operates within the process scope of the MySQL server and uses an internal API execute SQL as a certain MySQL user.

In general, a web service build atop of MySQL and the HTTP plugin can achieve similar levels of security, because they use a similar two-staged security approach. However, please note, the early development stage of the HTTP plugin for MySQL. It does not match the maturity and feature set web services yet.

Please, be aware that HTTP basic authentication allows a username and password to be given as part of the URL. As a result, the user credentials may appear in log files, including the history of a web browser. Future versions may offer more sophisticated HTTP authentication methods.

2.3 User concept

The user and security concept of the HTTP plugin is two-staged. First, clients use HTTP basic authentication to login towards the HTTP plugin itself. Then, if the plugin executes SQL statements to answer the HTTP request, the SQL statements are issues on behalf on a certain MySQL user. Which MySQL user depends on whether SSL/HTTPS is used or not.

Every resource is the HTTP plugin is protected using HTTP basic authentication. The use of HTTP basic authentication is mandatory. It cannot be disabled. All HTTP plugin server resources are protected with the same credentials. The HTTP basic authentication user name and passwort are configured through the server variables `myhttp_basic_auth_user_name` and `myhttp_basic_auth_user_passwd`.

If no SSL is used for a request and the HTTP plugin seeks to execute SQL after successful basic authentication, the plugin logs in towards MySQL as a certain MySQL user. MySQL identifies users by a host, a user name and a password. The host, user name and password of the MySQL user to be used by the HTTP plugin are configured with the server variables `myhttp_default_mysql_user_host`, `myhttp_default_mysql_user_name`, `myhttp_default_mysql_user_passwd`. This way, MySQL user credentials are never exposed in unprotected HTTP communication. The unencrypted HTTP communication only shows the HTTP basic authentication credentials used to authorize a client towards the HTTP plugin itself.

To execute SQL through the HTTP plugin as a different user but the predefined one, you must enable SSL support and issue a HTTPS request. In case of HTTPS the plugin uses the user name and password provided through HTTP basic authentication and the host from the HTTP request to log in towards MySQL. Here, basic authentication is only a vehicle to provide MySQL credentials. The level of security is

comparable to a standard client connection made through the MySQL Client/Server Protocol when using native authentication. If, and only if, a MySQL Client Server protocol connection is encrypted and protected using SSL, the password is sent from the client to the server without further encryption.

Currently, it is not possible to use SSL and restrict access to certain MySQL users. If SSL is enabled, all MySQL users can log in. It is desired to further develop the user concept to give more control.

Only MySQL users that use the MySQL native password encryption can be used. This is true for the default user of non-SSL connections and all other users in case of SSL connections.

2.4 Thread model

The thread model used for handling HTTP clients is more complex than the model used to handle standard MySQL Client/Server Protocol clients. MySQL handles standard client connections using one thread per connection. The thread handles the I/O operations and executes the clients commands. The HTTP Plugin uses distinct threads for handling client connections and SQL execution. A HTTP Plugin thread handing I/O operates indedently of the thread used for running SQL commands.

When the HTTP plugin is loaded into the MySQL server, the HTTP library starts threads to handle connections. Upon connection of a HTTP client, the HTTP library invokes methods of the plugins core module. The core module then may decide to create a MySQL thread for executing SQL statements. The MySQL thread is initialized and prepared for SQL execution. Then SQL statements are executed and results are sent back to the client through the HTTPs' library I/O thread. When all results have been sent, the MySQL thread is destroyed.

The separation of the connection handling from the SQL execution impacts pluggable authentication used by standard MySQL clients, the query cache and the commercial thread pool. All these MySQL server modules require some connectivity between the server module and the client through a certain network I/O abstraction layer. Because HTTP clients are handled using a HTTP library, the standard network I/O abstraction layer is not available to these server modules. This affects the following server modules:

- MySQL pluggable authentication supports authentication methods that use a handshake protocol to exchange passwords and other information. MySQL authentication plugins use an I/O API for exchanging messages that is not provided by the HTTP library.
- The MySQL query cache is acessing functionality from the I/O abstraction layer used for standard connections. Because this I/O abstraction layer is not available with HTTP connections, the HTTP plugin explicitly disables the query cache when executing SQL.
- The commercial thread pool plugin requires the use of the standard connections the I/O abstraction layer.

Chapter 3 Installation

Table of Contents

The HTTP Plugin is provided as a MySQL Labs release. Lab releases provide access to early development versions.

The Lab release consists of two parts. A special version of MySQL 5.7 and the HTTP Plugin itself. The plugin can be loaded into the special version of MySQL 5.7 that comes with the plugin only. Only this patched MySQL server provides the internal plugin APIs required.

3.1 Plugin Installation

Follow the standard MySQL server binary installation instructions to install the server. Load the plugin into the MySQL Server. Use `SHOW PLUGINS` to check if it has been loaded by MySQL.

```
mysql> INSTALL PLUGIN myhttp SONAME 'libmyhttp.so'
mysql> SHOW PLUGINS
...
mysql> SELECT * FROM INFORMATION_SCHEMA.PLUGINS WHERE PLUGIN_NAME='myhttp'\G
*************************** 1. row ***************************
PLUGIN_NAME: myhttp
PLUGIN_VERSION: 1.0
PLUGIN_STATUS: ACTIVE
PLUGIN_TYPE: DAEMON
PLUGIN_TYPE_VERSION: 50705.0
PLUGIN_LIBRARY: libmyhttp.so
PLUGIN_LIBRARY_VERSION: 1.5
PLUGIN_AUTHOR: Andrey Hristov, Ulf Wendel
PLUGIN_DESCRIPTION: HTTP Plugin for MySQL
PLUGIN_LICENSE: GPL
LOAD_OPTION: ON
1 row in set (0,09 sec)
```

After installation, the plugin will immediately start listening for HTTP client requests and begin handling them. The default HTTP listening port is `8080`. By default, HTTP basic authentication is used to secure accesses. The default HTTP basic authentication username is `basic_auth_user` and the passwort is `basic_auth_passwd`. You can issue a corresponding HTTP request to verify the HTTP server is running.

```
shell> curl --user basic_auth_user:basic_auth_passwd --url "http://127.0.0.1:8080/"

{
  "error": 404,
  "message": "Not Found"
}
```

3.2 Example setup and data

Most examples given in the manual assume some non-default settings. For example, the SQL user used by the examples is not created during a standard MySQL server installation. The user used for the examples will be granted access to the example database `myhttp` only. This is done to reduce the risk of accidentially exposing sensitive data through the HTTP interface.

The following variables should be added to the MySQL servers configuration file in the section `mysqld`. The listing shows all server variables available to configure the HTTP plugin.

```
#
# Default database, if no database given in URL
#
myhttp_default_db               = myhttp

#
# Non-SSL default MySQL SQL user
#
myhttp_default_mysql_user_name   = http_sql_user
myhttp_default_mysql_user_passwd = sql_secret
myhttp_default_mysql_user_host   = 127.0.0.1

# Change only, if need be to run the examples!
#
# General settings
#
# myhttp_http_enabled             = 1
# myhttp_http_port                = 8080
# myhttp_crud_url_prefix          = /crud/
# myhttp_document_url_prefix      = /doc/
# myhttp_sql_url_prefix           = /sql/
#
# Non-SSL HTTP basic authentication
#
# myhttp_basic_auth_user_name     = basic_auth_user
# myhttp_basic_auth_user_passwd   = basic_auth_passwd
#
# SSL
#
# myhttp_https_enabled            = 1
# myhttp_https_port               = 8081
# myhttp_https_ssl_key            = /path/to/mysql/lib/plugin/myhttp_sslkey.pem
```

Change the plugins server variables defaults for `myhttp_default_db` to `myhttp`. This is the name of the database (schema) that will be accessed, if no database name can be derived from the URL. The database `myhttp` will be created in the next step.

Set the default MySQL user for non-SSL connections with `myhttp_default_mysql_user_name`, `myhttp_default_mysql_user_passwd` and `myhttp_default_mysql_user_host`. All HTTP connections not using SSL and causing any SQL to be executed will use this MySQL user in. The user gets created in the next step and is given access to the `myhttp` database only.

The variables that are commented out do not need to be set unless, your setup requires changes. If, for example, you have already a network server bound to port 8080, then instruct the plugin to bind to a different port. After you have made the necessary changes, restart the MySQL server to make the changes take effect.

Create a test database with the name `myhttp`. Make sure the name matches the server variable `myhttp_default_db`.

```
DROP DATABASE IF EXISTS `myhttp`;
CREATE DATABASE `myhttp`;
```

Use the MySQL prompt to create the follwing tables and to insert some data into them. The tables will be queried by the examples in the manual.

```
USE `myhttp`;
CREATE TABLE `simple` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `col_a` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
INSERT INTO `simple` VALUES (1,'Hello'),(2,' '),(3,'world!');
CREATE TABLE `sql_types` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `col_char` char(127) NOT NULL,
  `col_null` char(1) DEFAULT NULL,
  `col_date` date NOT NULL,
  `col_decimal` decimal(5,2) NOT NULL,
  `col_float` float NOT NULL,
  `col_bigint` bigint(20) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
INSERT INTO `sql_types` VALUES (1,'CHAR(127)',NULL,'2014-08-21',
  123.45,0.9999,9223372036854775807);
INSERT INTO `sql_types` VALUES (2,'CHAR(127)',NULL,'2014-08-22',
  678, -1.11,-9223372036854775800);
CREATE TABLE `dojo_jsonp` (
  `modified` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  `version` int(10) unsigned DEFAULT '1',
  `dojo_blob` blob,
  `dojo_id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`dojo_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
INSERT INTO `dojo_jsonp` VALUES ('2014-08-22 07:11:43',1,'{\"first_name\": \"Ulf\",
  \"last_name\": \"Wendel\", \"email\": \"ulf.wendel@example.com\"}',1),
  ('2014-08-22 07:12:04',1,'{\"first_name\": \"Andrey\", \"last_name\": \"Hristov\",
  \"email\": \"andrey.hristov@example.com\"}',2);
CREATE TABLE `dojo_jsonp_fields` (
`first_name` varchar(255) DEFAULT NULL,
`last_name` varchar(255) DEFAULT '',
`email` varchar(255) DEFAULT '',
`modified` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
`version` int(10) unsigned DEFAULT '1',
`dojo_id` int(10) unsigned NOT NULL AUTO_INCREMENT,
PRIMARY KEY (`dojo_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
INSERT INTO `dojo_jsonp_fields` VALUES ('Andrey','Hristov',
  'andrey.hristov@example.com','2014-08-22 07:27:23',1,1);
INSERT INTO `dojo_jsonp_fields` VALUES ('Ulf','Wendel',
  'ulf.wendel@example.com','2014-08-22 07:30:37',1,2);
CREATE TABLE `no_primary_key` (
  `id` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
INSERT INTO `no_primary_key` VALUES (1),(2),(3);
CREATE TABLE `compound_primary_key` (
  `col_a` int(11) NOT NULL,
  `col_b` int(11) NOT NULL,
  PRIMARY KEY (`col_a`,`col_b`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
INSERT INTO `compound_primary_key` VALUES (1,1),(1,2),(1,3),(2,1);
```

Create the default SQL user of the HTTP plugin. The users password must be set using the servers `mysql_native_password` authentication plugin. No other authentication plugins are supported. The `mysql_native_password` is the default authentication plugin used by MySQL 5.7. To avoid any pitfalls with non-standard default server settings, the full SQL syntax is used to set the users password. Grant the user access to the `myhttp` database. Most examples use that database only.

```
CREATE USER 'http_sql_user'@'127.0.0.1' IDENTIFIED WITH mysql_native_password;
SET old_passwords = 0;
SET PASSWORD FOR 'http_sql_user'@'127.0.0.1' = PASSWORD('sql_secret');
```

```
GRANT ALL ON myhttp.* TO  'http_sql_user'@'127.0.0.1';
```

With the settings in place and the SQL objects created, verify the correctness of the setup. Execute `SELECT * FROM simple` on database `myhttp` through the `sql/` HTTP endpoint. Use a HTTP client to sent a HTTP 1.1 `GET` request for `/sql/myhttp/SELECT+%2A+FROM+simple` to the plugin listening to port `8080` on host `127.0.0.1`. Let the HTTP client authenticate itself through HTTP basic authentication using the username `basic_auth_user` and `basic_auth_passwd`. A convenient way of providing basic authentication information to the HTTP server is adding the credentials to the requested URL. You can use `curl` or a web browser sent the appropriate `GET` request by fetching the URL `http://basic_auth_user:basic_auth_passwd@127.0.0.1:8080/sql/myhttp/SELECT+%2A+FROM+simple`. Upon success, the HTTP plugin replies with the query result in JSON format.

```
shell> curl --user basic_auth_user:basic_auth_passwd
 --url "http://127.0.0.1:8080/sql/myhttp/SELECT+%2A+FROM+simple"

[
  {
    "meta": [
      {
        "type": 3,
        "catalog": "def",
        "database": "myhttp",
        "table": "simple",
        "org_table": "simple",
        "column": "id",
        "org_column": "id",
        "charset": 63,
        "length": 11,
        "flags": 16899,
        "decimals": 0
      },
      {
        "type": 253,
        "catalog": "def",
        "database": "myhttp",
        "table": "simple",
        "org_table": "simple",
        "column": "col_a",
        "org_column": "col_a",
        "charset": 33,
        "length": 765,
        "flags": 0,
        "decimals": 0
      }
    ],
    "data": [
      [
        "1",
        "Hello"
      ],
      [
        "2",
        " "
      ],
      [
        "3",
        "world!"
      ]
    ],
    "status": [
      {
        "server_status": 34,
        "warning_count": 0
```

```
      }
    ]
  }
]
```

3.3 Configuration server variables

The plugin is configured through server variables. The following variables exist.

```
mysql> SHOW VARIABLES LIKE 'myhttp%'

+---------------------------------+-----------------------------+
| Variable_name                   | Value                       |
+---------------------------------+-----------------------------+
| myhttp_basic_auth_user_name     | basic_auth_user             |
| myhttp_basic_auth_user_passwd   | basic_auth_passwd           |
| myhttp_crud_url_prefix          | /crud/                      |
| myhttp_default_db               | myhttp                      |
| myhttp_default_mysql_user_host  | localhost                   |
| myhttp_default_mysql_user_name  | http_sql_user               |
| myhttp_default_mysql_user_passwd | sql_secret                  |
| myhttp_document_url_prefix      | /doc/                       |
| myhttp_http_enabled             | ON                          |
| myhttp_http_port                | 8080                        |
| myhttp_https_enabled            | OFF                         |
| myhttp_https_port               | 8081                        |
| myhttp_https_ssl_key_file       | lib/plugin/myhttp_sslkey.pem |
| myhttp_sql_url_prefix           | /sql/                       |
+---------------------------------+-----------------------------+
14 rows in set (0,03 sec)
```

Name	Cmd-Line	Option file	System Var	Var Scope	Dynamic
myhttp_basic_auth_user_name	Yes	Yes	Yes	Global	No
myhttp_basic_auth_user_passwd	Yes	Yes	Yes	Global	No
myhttp_default_db	Yes	Yes	Yes	Global	No
myhttp_default_mysql_user_host	Yes	Yes	Yes	Global	No
myhttp_default_mysql_user_name	Yes	Yes	Yes	Global	No
myhttp_default_mysql_user_passwd	Yes	Yes	Yes	Global	No
myhttp_http_enabled	Yes	Yes	Yes	Global	No
myhttp_http_port	Yes	Yes	Yes	Global	No
myhttp_https_enabled	Yes	Yes	Yes	Global	No
myhttp_https_port	Yes	Yes	Yes	Global	No
myhttp_https_ssl_key_file	Yes	Yes	Yes	Global	No
myhttp_sql_url_prefix	Yes	Yes	Yes	Global	No
myhttp_crud_url_prefix	Yes	Yes	Yes	Global	No

myhttp_basic_auth_user_name

What	Description
Introduced	1.0

What	Description
Enpoints	all
System Variable Name	myhttp_basic_auth_user_name
Variable Scope	Global
Dynamic Variable	No
Type	string
Default	basic_auth_user

Login user name for HTTP basic authentication. Basic authentication cannot be disabled by configuring an empty user name.

Please note, clients can provide the HTTP basic authentication user name and passwort as part of the URL. This bares the risk of exposing user credentials in log files of any system involved in the processing of the HTTP request. If user credentials are encoded in the URL, they may appear in a clients browser history or the log files of a proxy server operating between the client and the server.

myhttp_basic_auth_user_passwd

What	Description
Introduced	1.0
Endpoints	all
System Variable Name	myhttp_basic_auth_user_passwd
Variable Scope	Global
Dynamic Variable	No
Type	string
Default	basic_auth_passwd

Login user password for HTTP basic authentication.

myhttp_default_db

What	Description
Introduced	1.0
Enpoints	all
System Variable Name	myhttp_default_db
Variable Scope	Global
Dynamic Variable	No
Type	string
Default	test

Default database/schema to use if no other selected.

myhttp_default_mysql_user_host

What	Description
Introduced	1.0

What	Description
Enpoints	all
System Variable Name	myhttp_default_mysql_user_host
Variable Scope	Global
Dynamic Variable	No
Type	string
Default	127.0.0.1

SQL account used to perform SQL queries for non-SSL connections. This SQL account determines the SQL access permissions of the HTTP interface user. The MySQL SQL account used is described with the three variables: `myhttp_default_mysql_user_host`, `myhttp_default_mysql_user_user`, `myhttp_default_mysql_user_passwd`.

The login process for the HTTP Plugin is two-staged. The procedure used for SSL and non-SSL connections differ. For a non-SSL connection, the client first uses HTTP Basic Authentication to authenticate itself against the HTTP Plugin. The HTTP Basic Authentication credentials are set through `myhttp_basic_auth_user_name`, `myhttp_basic_auth_user_passwd`. Then, after successful HTTP Basic Authentication, the HTTP Plugin executes SQL as the user set.

Standard MySQL Client/Server Protocol connections evaluate the clients host. The HTTP Plugin does not considered the host of the requesting client. Instead, the variable value is used.

myhttp_default_mysql_user_user

What	Description
Introduced	1.0
Enpoints	all
System Variable Name	myhttp_default_mysql_user_user
Variable Scope	Global
Dynamic Variable	No
Type	string
Default	root

Please, see also `myhttp_default_mysql_user_host`.

myhttp_default_mysql_user_passwd

What	Description
Introduced	1.0
Enpoints	all
System Variable Name	myhttp_default_mysql_user_passwd
Variable Scope	Global
Dynamic Variable	No
Type	string
Default	

Please, see also `myhttp_default_mysql_user_host`.

myhttp_http_enabled

What	Description
Introduced	1.0
Enpoints	all
System Variable Name	myhttp_http_enabled
Variable Scope	Global
Dynamic Variable	No
Type	boolean
Default	ON

Whether to enable listening for client requests on an HTTP port. Please, see also: `myhttp_https_enabled`.

myhttp_http_port

What	Description
Introduced	1.0
Enpoints	all
System Variable Name	myhttp_http_port
Variable Scope	Global
Dynamic Variable	No
Type	Numeric
Default	8080
Range	1 . . 65535

Network port to listen on for HTTP requests.

myhttp_https_enabled

What	Description
Introduced	1.0
Enpoints	all
System Variable Name	myhttp_https_enabled
Variable Scope	Global
Dynamic Variable	No
Type	boolean
Default	ON

Please note, the Lab release does not include SSL support for packaging reasons. SSL support is currently only available when using OpenSSL. This release does not include OpenSSL support.

Whether to enable listening for client requests on an HTTPS (SSL) port. Requires `myhttp_https_ssl_key_file` to be set properly. Please, see also: `myhttp_http_enabled`..

myhttp_https_port

What	Description
Introduced	1.0
Enpoints	all
System Variable Name	myhttp_https_port
Variable Scope	Global
Dynamic Variable	No
Type	Numeric
Default	8081
Range	1 . . 65535

Network port to listen on for HTTPS (SSL) requests.

myhttp_https_ssl_key_file

What	Description
Introduced	1.0
Enpoints	all
System Variable Name	myhttp_https_ssl_key_file
Variable Scope	Global
Dynamic Variable	No
Type	String
Default	lib/plugin/myhttp_sslkey.pem

SSL key file to use for HTTPS (SSL) connections.

myhttp_sql_url_prefix

What	Description
Introduced	1.0
Enpoints	sql
System Variable Name	myhttp_sql_url_prefix
Variable Scope	Global
Dynamic Variable	No
Type	String
Default	/sql/

URL prefix used for SQL endpoint (user API).

myhttp_crud_url_prefix

What	Description
Introduced	1.0

What	Description
Enpoints	crud
System Variable Name	myhttp_crud_url_prefix
Variable Scope	Global
Dynamic Variable	No
Type	String
Default	/crud/

URL prefix of the CRUD (Create-Read-Update-Delete) endpoint.

myhttp_document_url_prefix

What	Description
Introduced	1.0
Enpoints	Document
System Variable Name	myhttp_document_url_prefix
Variable Scope	Global
Dynamic Variable	No
Type	String
Default	/doc/

URL prefix of the JSON Document (DOC) endpoint.

Chapter 4 User APIs (endpoints)

Table of Contents

The HTTP plugin offers three HTTP endpoints, implementing thee different user APIs. The endpoints serve different use cases, offer different advantages and disadvantages. The following table gives an overview.

4.1 Feature comparison

Feature	**Endpoint**	**Endpoint**	**Endpoint**
Endpoint Name	SQL	CRUD	JSON Document
URL prefix	http[s]://server:port/sql/	http[s]://server:port/crud/	http[s]://server:port/doc/
Query capabilities	Rich: full power of SQL	Limited: primary key based CRUD (Create-Read-Update-Delete) to SQL tables	Limited: primary key based CRUD for JSON documents in SQL tables
Direct SQL	Yes	No	No
HTTP methods	GET	GET, PUT, DELETE	GET, PUT, DELETE
Transactions	Yes, autocommit	Yes, autocommit	Yes, autocommit
Charsets	utf8 only	utf8 only	utf8 only

Feature	Endpoint	Endpoint	Endpoint
Resultset meta data	Yes, full MySQL meta data	No	No. Plugin managed document revisions for optimistic locking.
HTTP Basic Auth	Yes	Yes	Yes

The SQL endpoint is the most powerful. It allows executing SQL over HTTP(S). Almost all queries a standard MySQL Connector can run over the MySQL Client/Server Protocol, can also be run over HTTP. A standard client connects to MySQL using the properitary, binary MySQL Protocol. The MySQL Client/Server Protocol is stateful and supports the concept of a session. A client connects, establishes a connection state and runs one or more commands before it disconnects. HTTP is stateless. There is no concept of a session. SQL features that require a session are not available with the HTTP SQL endpoint. Examples of such SQL features are transactions consisting of more than one command or prepared statements. Note, however, that the autocommit mode can be used. When using autocommit, a transaction consists of one command only.

To match the functionality of a standard client, the SQL endpoint is the only one to return full result set meta data. The result set meta data contains information such as the MySQL data types of results. Other endpoints skip this information for brevity.

The SQL endpoint is particularily useful for JavaScript clients that need the full power of SQL, or in environments where a firewall prevents the use of the properitary MySQL Client/Server Protocol and standard MySQL Connectors.

A simple primary key based access to SQL tables is offered by the CRUD (Create-Read-Update-Delete) endpoint. The CRUD endpoint is RESTful in the sense that rows are created using the HTTP `PUT` method, read using `GET`, updated with `PUT` and removed using the HTTP `DELETE` method. The query capabilities are limited to basic key value semantics. Results are returned as flat JSON documents. Results do not include result set meta data.

The CRUD interface can be used for simplistic spreadsheets where users search by primary key only. It may also be of use in heterogenous environments where databases are only used as a secure intermediate storage platform for data exchange. The availability of a simple to use API combined with a protocol that is allowed by many firewalls, may outweight the additional power offered by a standard MySQL client. An example of such an environment is a publishing house that manages articles in various systems and exports them to MySQL. Then, various online systems transform the articles and fill caches and webservers to publish them.

The document endpoint bends MySQL towards the key document data model. Similar to the CRUD endpoint accesses are mapped to tables based on their primary keys. The tables have a BLOB column that holds the document. The value stored in the BLOB column is assumed to be a valid JSON document. The document is not constrained in any way but that it must fit the SQL column type used to store it.

This endpoint emulates a key document store using MySQL and HTTP. Key document models are great if the documents structure changes frequently and thus cannot be easily mapped to a fixed, predefined set of columns. Documents offer a level schema flexibility that MySQL otherwise cannot achieve.

Furthermore documents form logical entities that qualify for sharding. An application using the key document model likely runs the majority of queries on individual documents only. Thus, distributed queries, which are expensive in any distributed database, are avoided.

4.2 Common properties

All endpoints share the following common properties and limitations.

- HTTP is a stateless protocol. The protocol does not include the concept of a session. None of the endpoints tries to overcome this property.
- Other charsets but UTF-8 are not supported. All results are encoded using UTF-8 (`utf8_general_ci`). Characters outside of the printable ASCII range or requiring multi-byte encoding may be returned using JSON `\uHEX` notation.

  Due to the escaping and resulting growth in content lengths, none of the SQL endpoints is ideal for handing binary data.
- Authentication and authorization pattern are independent of the SQL endpoint. All endpoints follow the same logic. All offer the same features.
- Endpoints are addressed with an URL prefix. The URL prefix is always followed by the name of the default database to use used. The URL pattern always begins with: `http[s]://host:port/endpoint_prefix/default_database/`. The pattern then continues with endpoint specific elements: `http[s]://host:port/endpoint_prefix/default_database/endpoint_specific_part`. The `default_database` is a mandatory element in the URL pattern of any endpoint. However, it is valid to use an empty string for the default database, for example, `http://127.0.0.1:8080/sql//SELECT+DATABSE()` is a valid URL of the SQL endpoint. If an emptry string is given, the plugin defaults to using the default database specified with the server variable `myhttp_default_db`.

4.3 The SQL endpoint: /sql/

The `sql` endpoint executes SQL statements "as-is". Arbitrary SQL commands are accepted by the endpoint. SQL commands are not validated or sanitized in any way. Therefore, database administrators should restrict access to this endpoint. Restricting access is good practice for any kind of clients using any protocol.

Most SQL statements that can be executed using a standard MySQL client can also be executed through the HTTP Plugins SQL endpoint. The use of the HTTP protocol causes some limitations. Below is a list of supported statements. The list may not be complete:

- `SELECT`
- Assorted `CREATE`, for example but not only, `CREATE TABLE`
- `UPDATE`, `REPLACE`, `INSERT`, `DELETE`
- `CALL` for functions and procedures. Including procedures that return multiple result sets.
- Specialized statements such as: `DO`, `HANDLER`.
- Administrative statements such as: `CREATE USER`.

HTTP is a stateless protocol. The protocol does not include the concept of a session. A SQL feature that requires session semantics cannot be used with the SQL endpoint. Corresponding SQL statements are not blocked. Their execution has the same effect as connecting to MySQL using a standard client, running a single SQL statement and disconnecting.

Some MySQL specific features also need a stateful message exchange between the client and the server. Because the SQL endpoint follows the stateless nature of HTTP and offers no session concept, they cannot be used either:

- SQL transactions with more than one command cannot be used because there is no way to send a sequence of commands using multiple HTTP requests. All SQL is executed in autocommit mode.

- The `SET` command can be used but you should use it for global settings only. Using it for the current session is questionable because the session ends with the execution of the `SET` command. For example, when you use the SQL endpoint to set a SQL session variable with `SET @myvar='hello'`, you will not be able to access the variable from the next HTTP request.
- The HTTP SQL endpoint does not support prepared statements. Prepared statements prepare a statement once, then allow multiple executions. For this, a client sends a statement to the server to be prepared. The server replies with a handle for the prepared statement. Then, the client sends parameter values for bound parameters, if any, before the statement gets executed. This prepared statement protocol requires session support, which is not available.

  The HTTP Plugin does not block the SQL syntax for prepared statement. It is possible to prepare a statement using `PREPARE`. But because the SQL endpoint lets you run only one command per HTTP request and the SQL session ends with the HTTP request, the prepared statement will not be accessible. A prepared statement created in one session is not available to other sessions.
- Most MySQL authentication plugins use a multi-message handshake protocol and cannot be used therefore. MySQL users used with the SQL plugin must be created using the MySQL native authentication plugin.

The above lists do not aim to be complete.

Other charsets but UTF-8 are not supported. All results are encoded using UTF-8 (`utf8_general_ci`). Characters outside of the printable ASCII range or requiring multi-byte encoding may be returned using JSON `\uHEX` notation.

Results include meta data information. Unlike some other interfaces, the HTTP SQL endpoint does not align or convert meta data information provided by the server. Here, the SQL endpoint behaves similar to the MySQL C API. Both interfaces provide the user with raw data from the server. It is then up to the user, for example, to handle differences between server versions, if any. Given that the plugin supports MySQL 5.7 only this should be no issue.

4.3.1 API overview

The SQL endpoint handles all HTTP(S) GET requests on the listening port and host of the HTTP Plugin with the URL prefix matching the server variable `myhttp_sql_url_prefix`. For example, when using server variable defaults, the SQL endpoint listens to all HTTP requests with an URL that begins with `http://127.0.0.1:8080/sql/` respectively `https://127.0.0.1:8081/sql/` for SSL requests. The full URL pattern for requests is `protocol://host:port/sql/database/statement_to_execute` (assuming `myhttp_sql_url_prefix = /sql/`). Only HTTP GET method requests following this pattern are understood and allowed.

The default database for SQL command execution is encoded in the URL: `protocol://host:port/sql/database/`. The database part is mandatory. It cannot be omitted but it is valid to pass an empty string: `protocol://host:port/sql//`. If an empty string is given, the server variable `myhttp_default_db` determines the default database. For example, `http://127.0.0.1:8080/sql/tests/` sets the default database to `test`. Whereas the URL `https://127.0.0.1:8081/sql//` contains an empty database and defaults to `myhttp_default_db`.

The SQL command to be executed is encoded in the URL of a GET request. To run a SQL command, URL encode the command and append it to the URL prefix of the `sql` endpoint. Assuming default settings, given the URL prefix `http://127.0.0.1:8080/sql/database/` and the SQL command `SELECT 1`, the resulting URL for query execution by the SQL endpoint is `http://127.0.0.1:8080/sql/database/SELECT+1`. Issuing a HTTP request for that URL will give a reply similar to:

```
shell> curl --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/sql/myhttp/SELECT+1"

[
{
"meta":[
        {"type":8,"catalog":"def","database":"","table":"","org_table":"",
        "column":"1","org_column":"","charset":63,"length":1,
        "flags":129,"decimals":0}
],
"data":[
        ["1"]
],
"status":[{"server_status":2,"warning_count":0}]
}
]
```

Please note, no pretty printing is done by the HTTP plugin. In the following the manual may make use of a pretty printer to improve readability.

It is not an error that the `database` meta data entry reports an empty string. This is the very same meta data that a standard client gets when executing the same statement. Should you have setup the example database, query the `simple` table from the database `myhttp` for an example returning multiple rows:

```
shell>   curl --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/sql//SELECT+%2A+FROM+simple+ORDER+BY+id"

[
  {
    "meta": [
      {
        "type": 3,
        "catalog": "def",
        "database": "myhttp",
        "table": "simple",
        "org_table": "simple",
        "column": "id",
        "org_column": "id",
        "charset": 63,
        "length": 11,
        "flags": 16899,
        "decimals": 0
      },
      {
        "type": 253,
        "catalog": "def",
        "database": "myhttp",
        "table": "simple",
        "org_table": "simple",
        "column": "col_a",
        "org_column": "col_a",
        "charset": 33,
        "length": 765,
        "flags": 0,
        "decimals": 0
      }
    ],
    "data": [
      [
        "1",
        "Hello"
      ],
      [
```

```
        "2",
        " "
      ],
      [
        "3",
        "world!"
      ]
    ],
    "status": [
      {
        "server_status": 2,
        "warning_count": 0
      }
    ]
  }
]
```

4.3.2 HTTP methods, headers and status codes

The SQL endpoint supports HTTP GET requests only. Other HTTP methods but GET are not allowed. The following HTTP headers are set for all HTTP GET replies.

Header	Description
Server	Always given. Can be considered as an API version. For example: `MyHTTP 1.0.0-alpha`
Cache-control	Always given. Always `must-revalidate`
Pragma	Always given. Always `no-cache`
Content-Length	Always given.
Content-Type	Set to `application/json` for `200 OK`, `400 Bad Request`, `401 Unauthorized`. Other replies may or may not contain it.
Connection	Always given. Always `Keep-Alive`.

Other HTTP methods but GET are not supported. The plugins replies to all other methods but GET by sending an empty reply with code `405 Method Not Allowed`.

```
shell> curl -v --request PATCH  --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/sql/myhttp/"

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> PATCH /sql/myhttp/ HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 405 Method Not Allowed
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 0
< Pragma: no-cache
```

Successful HTTP GET requests return status code `200 OK`.

```
shell> curl -v --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/sql/myhttp/SELECT+%27Good+old+SQL%27"

[...]
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 258
< Pragma: no-cache
< Content-Type: application/json
[
  {
    "meta": [
      {
        "type": 253,
        "catalog": "def",
        "database": "",
        "table": "",
        "org_table": "",
        "column": "Good old SQL",
        "org_column": "",
        "charset": 33,
        "length": 36,
        "flags": 1,
        "decimals": 31
      }
    ],
    "data": [
      [
        "Good old SQL"
      ]
    ],
    "status": [
      {
        "server_status": 2,
        "warning_count": 0
      }
    ]
  }
]
```

Unsuccessful requests, for example, failed SQL commands, return `400 Bad Request`.

```
shell> curl -v --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/sql/myhttp/NoSQL"

[...]
> GET /sql/myhttp/NoSQL HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 400 Bad Request
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 195
< Pragma: no-cache
< Content-Type: application/json
<
{
```

```
  "errno": 1064,
  "sqlstate": "42000",
  "error": "You have an error in your SQL syntax; check the manual that
  corresponds to your MySQL server version for the right syntax to use
  near 'NoSQL' at line 1"
}
```

If HTTP Basic Authentication fails, `401 Unauthorized` is sent.

```
shell> curl -v --url "http://wrong:user@127.0.0.1:8080/sql/myhttp/"

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'wrong'
> GET /sql/myhttp/ HTTP/1.1
> Authorization: Basic d3Jvbmc6dXNlcg==
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 61
< WWW-Authenticate: Basic realm="MySQL HTTP Server"
< Pragma: no-cache
< Content-Type: application/json
<
{
  "errno": 1045,
  "sqlstate": "28000",
  "error": "401 Unauthorized"
}
```

You should follow the rules for valid requests. Invalid requests may cause replies with no content or content that is not JSON. Applications should check the returned HTTP headers before attempting to parse a reply as JSON. Consumers should only attempt parsing if the HTTP reply code is 200, 400 or 401, the HTTP header `Content-Type: application/json` is set and a correct `Content-Length` is given. Below is an example of an HTTP request that is not understood by the SQL endpoint. Please note, that the `Content-Type` header is omitted for the code 404 reply.

```
shell> telnet 127.0.0.1 8080

Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET jdsjsdjd HTTP/1.1

HTTP/1.1 404 Not Found
Connection: Keep-Alive
Cache-control: must-revalidate
Server: MyHTTP 1.0.0-alpha
Content-Length: 36
Pragma: no-cache

{
  "error": 404,
  "message": "Not Found"
}
```

The HTTP Plugin may even reply with no JSON content at all to other requests it does not understand. This is a known limitation.

4.3.3 JSON with padding (JSONP)

JSON with padding (JSONP) is supported to overcome the same origin policy restriction of JavaScript scripts run in a Browser. Please see, Wikipedia or the JavaScript examples given below for an explanation of the JSONP concept.

Parameter	Description
jsonp=<callback>	Enables JSON with padding. The parameters value is used as the callback function name. Replies returned from the plugin change from `reply` to `callback(reply)`.
jsonp_escape	Only considered when `jsonp` is also given. Replies returned from the plugin change from `reply` to `callback("reply")` with `reply` being escaped appropriately.

Add a `jsonp` parameter to the request URL for JSONP formatting. The `jsonp` parameter value is the function name used in JSONP formatting. For example, to get a JSONP formatted result to invoke the JavaScript callback `myfunction` for the SQL command `SELECT col_a FROM simple ORDER BY id` use `http://basic_auth_user:basic_auth_passwd@127.0.0.1:8080/sql/myhttp/SELECT+col_a+FROM+simple+ORDER+BY+id?jsonp=myfunction`;

```
shell> curl --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/sql/myhttp/SELECT+col_a+FROM+simple+ORDER+BY+id?jsonp=myfunction"

myfunction([
{
"meta":[
   {"type":253,"catalog":"def","database":"myhttp","table":"simple",
   "org_table":"simple", "column":"col_a","org_column":"col_a","charset":33,
   "length":765,"flags":0,"decimals":0}
],
"data":[
  ["Hello"],
  [" "],
  ["world!"]
],
"status":[{"server_status":2,"warning_count":0}]
}
]);
```

By default, the HTTP plugin replies are passed as is to the callback function as an argument. This approach can be used for all valid requests. The HTTP plugin will return valid JSON in response to well-formed requests. JSON does not need to be passed as a string but can be passed as a JSON object to the callback function.

In error situations it may happen that the HTTP does not return valid JSON. Then, the callback function either gets no argument or an invalid JavaScript JSON object. To avoid problems with invalid replies, the HTTP plugin can be instructed to pass its reply to the callback function as a string. When doing so, invalid JSON does not cause an immediately parse error but can be handled gracefully.

To overcome problems with faulty replies and to serve JavaScript frameworks that expect strings instead of JSON objects, the HTTP plugin can be instructed to escape the JSON reply as a string. The string is then passed as the first and only argument to the callback function. Use of a string can be requested with the HTTP parameter `jsonp_escape`. The `jsonp_escape` parameter value is not checked.

```
shell> curl --user basic_auth_user:basic_auth_passwd
 --url "http://127.0.0.1:8080/sql/myhttp/SELECT+col_a+FROM+simple+WHERE+id+%3D+1?jsonp=myfunction&jsonp_escape

myfunction("[
{
\"meta\":[
  {\"type\":253,\"catalog\":\"def\",\"database\":\"myhttp\",\"table\":\"simple\",
  \"org_table\":\"simple\",\"column\":\"col_a\",\"org_column\":\"col_a\",
  \"charset\":33,\"length\":765,\"flags\":0,\"decimals\":0} ],
\"data\":[
  [\"Hello\"]
],
\"status\":[{\"server_status\":2,\"warning_count\":0}]
}
]");
```

4.3.4 JSON content formats

The HTTP plugins SQL endpoint either replies with no content or with a valid JSON document to valid user requests. A JSON document shall be any JSON array or object that conforms to the syntax outlined in ECMA-404. To our knowledge, the specifications of the JSON syntax found on json.org, from ECMA-262, from RFC4627 and from RFC7159 agree on syntactic elements of the language.

No content is returned in reply to requests that use any HTTP method but GET. Valid HTTP GET requests are replied to with an error document or a result set document.

All JSON replies are encoded as UTF-8. Special characters are encoded using JSONs `\\u four-hex-digits` Unicode notation. Below is an example showing how the german umlauts `äöüßÄÖÜ` are escaped:

```
shell> curl --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/sql/myhttp/SELECT+%27German+umlauts%3A+%C3%A4%C3%B6%C3%BC%C3%9F%C3%84%C3%96%C3%

[
  {
    "meta": [
      {
        "type": 253,
        "catalog": "def",
        "database": "",
        "table": "",
        "org_table": "",
        "column": "_json_unicode_notation",
        "org_column": "",
        "charset": 33,
        "length": 69,
        "flags": 1,
        "decimals": 31
      }
    ],
    "data": [
      [
        "German umlauts: \u00e4\u00f6\u00fc\u00df\u00c4\u00d6\u00dc"
      ]
    ],
    "status": [
      {
        "server_status": 2,
        "warning_count": 0
      }
    ]
  }
]
```

```
```

JSON with padding (JSONP) can be requested for all valid requests. The padding will be applied to both kinds of replies that can occur: JSON result documents and JSON error documents.

```
REPLY:
  jsonp_function(answer) |
  answer

jsonp_function:
string

answer:
  result_sets |
  server_status |
  error

result_sets:
[
  result_set (, result_set ... )
  (, server_status )
]
```

The clients SQL statement and the HTTP code returned from the server can be used to determine the expected kind of reply. The HTTP code `400 Bad Request` may be followed by an error document. The error document is not returned for successful requests and replies using the HTTP code `200 OK`. The content of an a `200 OK` OK reply from the server shows one or more result documents. There is one result document for each result set produced by the server. How many the server produces depends on the SQL statement executed.

HTTP return code	SQL statment executed	Contents to expect
200 OK	`SELECT`	Result document without server status
200 OK	`INSERT` or other data manipulation statements.	Result document consisting of server status only
200 OK	`CREATE` or other data definitions statements.	Result document consisting of server status only
200 OK	`CALL` or statements producing multiple result sets	List of result documents without server status followed by server status
400 Bad Request	Any	Error document
401 Unauthorized	Any	Error document

4.3.4.1 JSON result document

Valid GET requests to run SQL statements that can return rows cause a JSON result set object to be returned. The JSON result set object format is used regardless how many rows have been produced.

A result set object always has three top level members: `meta`, `data` and `status`. The `meta` member holds an array of column meta data objects. Row data is listed in the `data` array. The `status` member object contains `server_status` and `warning_count`.

```
result:
  result_sets |
  server_status |
  error
```

```
result_sets:
[
  result_set (, result_set ... )
  (, server_status )
]

result_set:
{
  "meta" : [ column_meta (, column_meta ...) ],
  "data" : [ data_row (, data_row ...) ],
  "status" : [ result_status (, result_status ...) ]
}

column_meta:
{
  "type" : int,
  "catalog" : string,
  "database" : string,
  "table" : string,
  "org_table" : string,
  "column" : string,
  "org_column" : string,
  "charset": int,
  "length": int,
  "flags": int,
  "decimals": int
}

data_row:
[
  "column_value" (, "column_value" ...)
]

column_value:
  string |
  null

result_status:
{
  "server_status" : int,
  "warning_count" : int
}

server_status:
{
  "server_status" : int,
  "warning_count" : int,
  "affected_rows" : int,
  "last_insert_id" : int
}
```

A basic `SELECT` such as `SELECT 1, 'two' FROM DUAL` returns a JSON document holding an array of resulset objects, assuming that the SQL execution does not fail. Because `SELECT` returns at most one resultset, the list holds only one resultset object.

```
[
  {
    "meta": [
      {
        "type": 8,
        "catalog": "def",
        "database": "",
        "table": "",
        "org_table": "",
        "column": "1",
```

```
        "org_column": "",
        "charset": 63,
        "length": 1,
        "flags": 129,
        "decimals": 0
      },
      {
        "type": 15,
        "catalog": "def",
        "database": "",
        "table": "",
        "org_table": "",
        "column": "two",
        "org_column": "",
        "charset": 83,
        "length": 9,
        "flags": 129,
        "decimals": 31
      }
    ],
    "data": [
      [
        "1",
        "two"
      ]
    ],
    "status": [
      {
        "server_status": 2,
        "warning_count": 0
      }
    ]
  }
]
```

For each row generated generated by the executed SQL commands, an array of column values is added to the `data` member of the result set object. All SQL values but SQL `NULL` are serialized as JSON strings. A SQL `NULL` is serialized as a JSON `NULL`. The conversion of any SQL type but NULL to string similar to a behaviour that can also be observed with standard MySQL clients that use the MySQL client server protocol. The MySQL Client/Server Protocol has two operational modes: the text protocol and the binary protocol. Any non-prepared statement defaults to the text protocol. When the text protocol is used, the server transfers all data as binary strings to the client. It is then the clients task to map the strings to native datatypes. This is usually done by the MySQL driver used. Drivers base their conversion logic on local domain rules and the meta data information the server provides.

The SQL endpoint of the HTTP plugin does not attempt to map SQL types to JSON types. The SQL type system knows more scalar data types than the JSON type system. For example, a MySQL temporal datatype such as `DATE` has no counterpart in the JSON type system. The scalar data types supported by JSON are `string`, `number`, `null`, `true` and `false`. Because the HTTP plugin does not know anything about the consumer of the JSON results and its requirements, no attempt is made to offer a more complex type mapping.

```
shell> curl  --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/sql/myhttp/SELECT+%2A+FROM+sql_types+ORDER+BY+id"

[
  {
    "meta": [
      {
        "type": 3,
        "catalog": "def",
```

```
      "database": "myhttp",
      "table": "sql_types",
      "org_table": "sql_types",
      "column": "id",
      "org_column": "id",
      "charset": 63,
      "length": 11,
      "flags": 16899,
      "decimals": 0
    },
    {
      "type": 254,
      "catalog": "def",
      "database": "myhttp",
      "table": "sql_types",
      "org_table": "sql_types",
      "column": "col_char",
      "org_column": "col_char",
      "charset": 33,
      "length": 381,
      "flags": 4097,
      "decimals": 0
    },
    {
      "type": 254,
      "catalog": "def",
      "database": "myhttp",
      "table": "sql_types",
      "org_table": "sql_types",
      "column": "col_null",
      "org_column": "col_null",
      "charset": 33,
      "length": 3,
      "flags": 0,
      "decimals": 0
    },
    {
      "type": 10,
      "catalog": "def",
      "database": "myhttp",
      "table": "sql_types",
      "org_table": "sql_types",
      "column": "col_date",
      "org_column": "col_date",
      "charset": 63,
      "length": 10,
      "flags": 4225,
      "decimals": 0
    },
    {
      "type": 246,
      "catalog": "def",
      "database": "myhttp",
      "table": "sql_types",
      "org_table": "sql_types",
      "column": "col_decimal",
      "org_column": "col_decimal",
      "charset": 63,
      "length": 7,
      "flags": 4097,
      "decimals": 2
    },
    {
      "type": 4,
      "catalog": "def",
      "database": "myhttp",
      "table": "sql_types",
```

```
        "org_table": "sql_types",
        "column": "col_float",
        "org_column": "col_float",
        "charset": 63,
        "length": 12,
        "flags": 4097,
        "decimals": 31
      },
      {
        "type": 8,
        "catalog": "def",
        "database": "myhttp",
        "table": "sql_types",
        "org_table": "sql_types",
        "column": "col_bigint",
        "org_column": "col_bigint",
        "charset": 63,
        "length": 20,
        "flags": 4097,
        "decimals": 0
      }
    ],
    "data": [
      [
        "1",
        "CHAR(127)",
        null,
        "2014-08-21",
        "123.45",
        "0.9999",
        "9223372036854775807"
      ],
      [
        "2",
        "CHAR(127)",
        null,
        "2014-08-22",
        "678.00",
        "-1.11",
        "-9223372036854775800"
      ]
    ],
    "status": [
      {
        "server_status": 2,
        "warning_count": 0
      }
    ]
  }
]
```


Each `result_set` object has three members: `meta`, `data`, `status`. The `meta` member contains a list of metadata objects which describe the columns in the resulsets data rows. The metadata objects has entries for all columns in the resultset. The column metadata entries order maps the resultset column order. The first column metadata entry refers to the first column in the resultset and so forth.

The following list describes the members of the `meta` object. Many members correspond to elements of the same name in the C-API data structure `MYSQL_ROW`.

- `int type`

SQL data type. Please note, standard MySQL Clients can encapsulate the type numbers using type constants. Encapsulation help whenever type numbers change. The HTTP Plugin Lab Release does expose the type numbers without any encapsulation.

Type	Description
0	MYSQL_TYPE_DECIMAL (DECIMAL or NUMERIC)
1	MYSQL_TYPE_TINY (TINYINT)
2	MYSQL_TYPE_SHORT (SMALLINT)
3	MYSQL_TYPE_LONG (INTEGER)
4	MYSQL_TYPE_FLOAT (FLOAT)
5	MYSQL_TYPE_DOUBLE (DOUBLE or REAL)
6	MYSQL_TYPE_NULL (NULL)
7	MYSQL_TYPE_TIMESTAMP (TIMESTAMP)
8	MYSQL_TYPE_LONGLONG (BIGINT)
9	MYSQL_TYPE_INT24 (MEDIUMINT)
10	MYSQL_TYPE_DATE (DATE)
11	MYSQL_TYPE_TIME (TIME)
12	MYSQL_TYPE_DATETIME (DATETIME)
13	MYSQL_TYPE_YEAR (YEAR)
14	MYSQL_TYPE_NEWDATE
15	MYSQL_TYPE_VARCHAR.This type is never reported. The MySQL server converts it into 253/MYSQL_TYPE_VAR_STRING (VARCHAR or VARBINARY) when a client is using the text variant of the MySQL Client/Server protocol (`COM_QUERY`). This conversion happens inside the server and normal clients using, for example, the C-API function `mysql_query()` will never see type MYSQL_TYPE_VARCHAR. To make testing of the plugin easier and to align behaviour of, the plugin does the same conversion: 15/MYSQL_TYPE_VARCHAR is always reported as 253/MYSQL_TYPE_VAR_STRING.
16	MYSQL_TYPE_BIT (BIT)
17	MYSQL_TYPE_TIMESTAMP2. This type should never be reported. It is used in the server only. Please, report a bug if you can observe it.
18	MYSQL_TYPE_DATETIME2. This type should never be reported. It is used in the server only. Please, report a bug if you can observe it.
19	MYSQL_TYPE_TIME2. This type should never be reported. It is used in the server only. Please, report a bug if you can observe it.
246	MYSQL_TYPE_NEWDECIMAL (Precision math DECIMAL or NUMERIC)
247	MYSQL_TYPE_ENUM (ENUM)
248	MYSQL_TYPE_SET (SET)
249	MYSQL_TYPE_TINY_BLOB
250	MYSQL_TYPE_MEDIUM_BLOB
251	MYSQL_TYPE_LONG_BLOB

Type	Description
252	MYSQL_TYPE_BLOB (BLOB or TEXT)
253	MYSQL_TYPE_VAR_STRING (VARCHAR or VARBINARY).
254	MYSQL_TYPE_STRING (CHAR or BINARY)
255	MYSQL_TYPE_GEOMETRY (Spatial field)

- `string catalog`

  The catalog name. This value is always "def".

- `string database`

  The name of the database that the field comes from. Please, see the `db` description of the C-API documentation for details.

- `string table`

  The name of the table containing this field, if it isn't a calculated field. Please, see the C-API documentation for details.

- `string org_table`

  The name of the table. Please, see the C-API documentation for details.

- `string column`

  The name of the field. If the field was given an alias with an `AS` clause, the value of `column` is the alias. Please, see the C-API documentation (`name`) for details.

- `string org_column`

  The name of the field. Aliases are ignored. Please, see the C-API documentation (`org_name`) for details.

- `int charset`

  An ID number that indicates the character set/collation pair for the field. Please, see the C-API documentation (`charsetnr`) for details.

- `int length`

  The width of the field. This corresponds to the display length, in bytes. Please, see the C-API documentation for details.

- `int flags`

  Bit-flags that describe the field. The flags value may have zero or more of the bits set that are shown in the following table. Please, see the C-API documentation for details.

- `int decimals`

  The number of decimals for numeric fields, and (as of MySQL 5.6.4) the fractional seconds precision for temporal fields.

The `data` member of the `result_set` is a list of the actual data rows returned. Each list entry is in turn an array holding the column values for the row. All results are returned as strings. No attempt is made to map SQL types to corresponding JSON data types.

The number of SQL warnings caused by the SQL command that has generated a resultset is reported through the `status` member of the `result_set` object. The `status` member is an object with two members: `server_status` and `warning_count`.

- `int server_status`

- `int warning_count`

  The number of SQL warnings caused by the SQL command.

Stored procedures can return more than one result set. An example of such a stored procedure is given below.

```
CREATE PROCEDURE c_proc() BEGIN SELECT 1; SELECT 2; END
```

When executed using `CALL`, the `sql` endpoint returns a JSON document with a list of two resultset objects followed by a server status object.

```
[
  {
    "meta": [
      {
        "type": 8,
        "catalog": "def",
        "database": "",
        "table": "",
        "org_table": "",
        "column": "1",
        "org_column": "",
        "charset": 63,
        "length": 1,
        "flags": 129,
        "decimals": 0
      }
    ],
    "data": [
      [
        "1"
      ]
    ],
    "status": [
      {
        "server_status": 10,
        "warning_count": 0
      }
    ]
  },
  {
    "meta": [
      {
        "type": 8,
        "catalog": "def",
        "database": "",
        "table": "",
        "org_table": "",
        "column": "2",
        "org_column": "",
        "charset": 63,
        "length": 1,
        "flags": 129,
```

```
        "decimals": 0
      }
    ],
    "data": [
      [
        "2"
      ]
    ],
    "status": [
      {
        "server_status": 10,
        "warning_count": 0
      }
    ]
  },
  {
    "server_status": 2,
    "warning_count": 0,
    "affected_rows": 0,
    "last_insert_id": 0
  }
]
```

The server status object has four members: `server_status`, `warning_count`, `affected_rows` and `last_insert_id`.

- `int server_status`

- `int warning_count`

  Number of SQL warnings caused by the execution of the SQL statement.

- `int affected_rows`

  Number of rows affected by the SQL statement. Please, see the C-API `mysql_affected_rows()` for details.

- `int last_insert_id`

  The `ID` generated for an `AUTO_INCREMENT` column by the previous query. Please, see the C-API `mysql_insert_id()` for details.

A server status object is also sent in reply to data manipulation statements (DML) and data definition statement (DDL). Examples for data manipulation statements are `INSERT` and `UPDATE`.

```
shell> curl --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/sql/myhttp/INSERT+INTO+simple%28col_a%29+VALUES+%28%27Yippie%27%29"

{
  "server_status": 2,
  "warning_count": 0,
  "affected_rows": 1,
  "last_insert_id": 5
}
```

`CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`, `CREATE INDEX` are examples of data definition statement. Below is the servers reply to the data definition statement `DROP TABLE IF EXISTS unknown`.

```
```

```
{
  "server_status": 2,
  "warning_count": 1,
  "affected_rows": 0,
  "last_insert_id": 0
}
```

4.3.4.2 JSON error document

In case of an error, the plugin returns a JSON error object.

```
error:
{
  "errno": int,
  "sqlstate" : string,
  "error": string
}
```

A JSON object describing and error will be returned in reply to failed SQL statements but may occur it other contexts too. The JSON error document shall be expected when the HTTP return code is either `400 Bad Request` or `401 Unauthorized`.

```
shell> curl -v --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/sql/myhttp/Yet+another+error"

[...]
< HTTP/1.1 400 Bad Request
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 207
< Pragma: no-cache
< Content-Type: application/json
<
{
  "errno": 1064,
  "sqlstate": "42000",
  "error": "You have an error in your SQL syntax; check the manual that
  corresponds to your MySQL server version for the right syntax to
  use near 'Yet another error' at line 1"
}
```

The error object has the following members.

- `int errno`

  An error code for the error. Errors also listed at Appendix C, Errors, Error Codes, and Common Problems.

- `string sqlstate`

  The SQL state code of the error.

- `string error`

  An error description.

4.3.5 JavaScript examples: basics, jQuery, Dojo

Various JavaScript usage examples are given in the `docs/JavaScript` directory contained in the source tree. The examples show how to use query the HTTP `sql` endpoint using plain JavaScript run in a browser and how to create a Dojo JavaScript framework data store for accessing MySQL.

4.3.5.1 Introduction and general notes

The following introduction applies to all endpoints and user APIs. It describes the different techniques available to issue HTTP requests with JavaScript scripts run in a browser and common beginners pitfalls.

Most browsers execute JavaScript programs in a sandbox. A sandbox is a secured environment that forbids potentially insecure actions. For example, local file system accesses can be considered a risk as they would allow a malicious JavaScript program, embedded in an HTML page, access the users file system, possibly without notice by the user. But also network requests are usually limited to prevent cross-site scripting attacks.

Until recently HTTP was the only network protocol supported by JavaScript. But MySQL uses the properitary MySQL client server protocol to communicate with standard SQL clients. Thus, JavaScript programs executed in a browser had no way to communicate with MySQL.

Client-side JavaScript had to make use of a proxy to communicate with MySQL. The proxy would speak HTTP with the client-side JavaScript program and translate it into appropriate SQL statements to be send to MySQL using any of the MySQL Connectors and the properitary MySQL Client/Server Protocol. Then, the proxy would translate replies from MySQL into something easy to parse for the JavaScript program and send a reply to the client-side JavaScript programs request. Most often script languages such as PHP, Perl or Python have been used for proxying tasks. In very simple words, the HTTP Plugin for MySQL now takes over this proxy task. By doing so, the entire software stack is simplified.

A majority of the recent browsers implement a `XMLHttpRequest` object which can be used to send HTTP requests and read replies. Attempts are being made to standarize this interface, see also http://www.w3.org/TR/XMLHttpRequest2/.

The `XMLHttpRequest` object can be used to query any of the MySQL HTTP endpoints. The below example shows an HTML file with embedded client-side JavaScript which issues a `SELECT 'Greetings!'` question using `myhttp` as a default database. It is assumed that MySQL runs on local host, listens on port 8080 and HTTP basic auth is configured with the username `basic_auth_user` and the password `basic_auth_passwd`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>MySQL HTTP SQL endpoint: XHR example</title>
</head>
<body>
  <script type="application/javascript">
  try {
    xmlHttp = new XMLHttpRequest();
    xmlHttp.open('GET', 'http://127.0.0.1:8080/sql/myhttp/SELECT%20\'Greetings!\'',
      true, 'basic_auth_user', 'basic_auth_passwd');
    xmlHttp.onreadystatechange = function () {
      document.write("Response: " + xmlHttp.responseText);
    };
    xmlHttp.send(null);
  } catch (e) {
    alert(e);
  }
```

```
  </script>
  <p>
    NOTE: It is perfectly valid to get an error! Ask yourself why and how to
    prevent. Or, check the next example...
  </p>
</body>
</html>
```

Upon loading the HTML file into a browser, the JavaScript program is run and the result of the MySQL request inserted into the HTML document. This approach is very simple but unfortunately, it cannot always be used. The above may fail to access MySQL. The browsers sandbox limits `XMLHttpRequest` to access resources with the same origin only. For example, if you used the file protocol (`file://path/to/above_script.html`) to load the script into your browser, it will not be allowed to access any resource on `http://127.0.0.1`. Whereas, if the above HTML document is served by a web server running on host `127.0.0.1` and the browser has loaded it using `http://127.0.0.1/path/to/above_script.html`, the script will be allowed to access MySQL also running on `127.0.0.1`.

By default, it is not possible to request HTTP resources from remote servers this way. Using this approach, cross origin requests are only possible with clients and servers that support cross-origin resource sharing (CORS). The Lab release of the HTTP plugin for MySQL does not support the CORS standard but an alternative method to overcome the same origin policy.

The browsers same origin policy can also be worked around using the HTML `script` tag. HTML pages can embed scripts from arbitrary servers. Most modern AJAX enabled websites use this approach to surpass the limitations of the `XMLHttpRequest` object.

The below HTML example demonstrates the idea. The HTML page loads a script from the URL `http://example.com`. Note that any URL can be specified. The webserver `http://example.com` then sends a valid JavaScript program as a reply. Once the reply has arrived, the browser executes the JavaScript program.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>MySQL HTTP SQL endpoint: JSONP example</title>
  <script type="application/javascript">
  function response(results) {
    document.write("<p>Response callback called...</p>");
    document.write("<pre>" + results + "</pre>");
    if ("data" in results[0]) {
      document.write("<pre>" + results[0].data[0] + "</pre>");
    }
    if ("error" in results) {
      alert("Something went wrong, MySQL reports " + results.error);
    }
  }
  </script>
  <script type="test/javascript"
    src="http://basic_auth_user:basic_auth_passwd@example.com:8080/sql/
    myhttp/SELECT%20'Greetings!'?jsonp=response"></script>
</head>
<body>
</body>
</html>
```

If the client-side JavaScript is accessing a remote data source it is common practice for the server to return a JavaScript program that contains nothing but one function call. The name of the callback function can be

choosen freely by the caller. Once the caller has received the servers reply, the program is executed by the browser and the callers callback function is run.

In case of the HTTP Plugin, the called function is given one argument. The argument contains a valid JSON document. The document is the actual payload, it contains the actual reply of the data source. Because JSON is padded with a function, this pattern is called JSONP.

```
shell> curl -v --user basic_auth_user:basic_auth_passwd
  --url "http://example.com:8080/sql/myhttp/SELECT+%27JSONP%27?jsonp=callback"

* Connected to example.com (example.com) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> GET /sql/myhttp/SELECT+%27JSONP%27?jsonp=callback HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: example.com:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 255
< Pragma: no-cache
< Content-Type: application/javascript
<
callback([
{
 "meta":[
  {"type":253,"catalog":"def","database":"","table":"","org_table":"",
  "column":"JSONP","org_column":"",
  "charset":33,"length":15,"flags":1,"decimals":31}
 ],
 "data":[
  ["JSONP"]
 ],
 "status":[{"server_status":2,"warning_count":0}]
}
]);
```

Please note, the communication between the browser and the webserver is asynchronous. Any concurrently running client-side JavaScript is not blocked while the browser loads the script in the background. On the contrary, if a client sends multiple asynchronous requests in row, there is no guarantee that the requests will be received and performed by the server in the order sent nor that replies arrive in order at the sender. This can be very confusing and lead to errors if you are used to synchronous program execution only. Imagine the two queries `UPDATE account SET balance = balance + 1` and `UPDATE account SET balance = balance * 2` are run at random order. Given `balance = 100` before execution of the two statements the outcome can either be `balance = (100 + 1) * 2 = 202` or `balance = (100 * 2) + 1 = 201`.

The HTTP protocol is stateless. There is no concept of a transaction or session that spawns multiple requests. Thus, all MySQL commands executed by the HTTP Plugin are run in autocommit mode. Each and every successful command commits. Abort or rollback are not possible.

4.3.5.2 JavaScript framework examples

Assorted frameworks for browser-side JavaScript exist. In the following, we show usage examples for some of them. Please understand, that we cannot cover all possible frameworks. Our selection covers only some randomly selected ones to give an impression of patterns commonly found.

The examples may be very short and hint only how to perform a JSONP request. Result processing and error handling is often omitted. Production code should never trust input from JSONP requests but validate it.

The discussion of the dojo framework is more complete. It does sketch the challenges of asynchronous SQL over HTTP, gives code for parsing replies, handling errors and explains the limits of using MySQL as a data store with the framework. Although the code is for dojo, it may be a good introduction to some concepts and pitfalls that can arise no matter what framework is used.

jQuery

The jQuery JavaScript framework is a small and lightweight solution. A JSONP request to the HTTP Plugin for MySQL can be made through the `ajax` object jQuery provides.

```
<!doctype html>
<html>
<head>
 <meta charset="utf-8" />
 <title>Simple JSONP request</title>
</head>
<body>
 <p>
 Example how to use <a href="http://jquery.com/">jQuery</a>. If needed,
 edit this file to load jQuery, set the HTTP Plugin connection parameter, and reload.
</p>
 <script src="http://code.jquery.com/jquery-2.1.1.js"></script>
 <script>
 $( document ).ready(function() {

  var ret = $.ajax({
   url: "http://127.0.0.1:8080/sql/myhttp/SELECT%20'Hello world!'%20FROM%20DUAL",
   type: "GET",
   username: "basic_auth_user",
   password: "basic_auth_passwd",
   dataType : "jsonp",
   jsonp: "jsonp",
   success: function( json ) {
    alert("Success, first data row: " + json[0].data[0]);
   },
   error: function( xhr, status, error_thrown ) {
    alert( "Sorry, there was a problem (did you setup things first?): " +
      status + "/" + error_thrown );
   },
  });
 });
 </script>
</body>
</html>
```

Dojo Toolkit

The Dojo Toolkit is a feature rich JavaScript framework for developing desktop and mobile device client-side JavaScript applications. It does not only offer basic and portable APIs for interfacing with the HTTP Plugin but also advanced data store abstractions. The data store abstractions in turn can be linked to many graphics and charts to visualize data.

In the following some aspects of using the Dojo Toolkit with MySQL are discussed in brevity and a quick overview on further examples contained in the source distribution of the HTTP Plugin are given. If you are unfamiliar with the ideas and basic usage pattern of the Dojo Toolkit, please consult the Dojo manual first.

The Dojo Toolkit abstracts some aspects of client-side JavaScript development that cannot be explained in an aside manner.

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="utf-8">
 <title>Dojo Toolkit JSONP request</title>
</head>
<body>
 <script src="http://ajax.googleapis.com/ajax/libs/dojo/1.10.0/dojo/dojo.js"
   data-dojo-config="async: true"></script>
 <script>
  require(["dojo", "dojo/io/script", "dojo/domReady!"],
   function(dojo) {
    function queryMySQL() {
     var targetNode = dojo.byId("results");
     var jsonpArgs = {
      url: "http://basic_auth_user:basic_auth_passwd@127.0.0.1:8080/sql/myhttp/SELECT%201%20FROM%20DUAL",
      callbackParamName: "jsonp",
      load: function(data) {
       targetNode.innerHTML = "<pre>" + dojo.toJson(data, true) + "</pre>";
      },
      error: function(error) {
       targetNode.innerHTML = "An unexpected error occurred: " + error;
      }
     };
     dojo.io.script.get(jsonpArgs);
    }
    dojo.ready(queryMySQL);
   }
  );
 </script>
 <div id="results"></div>
</body>
</html>
```

The above example performs a JSONP request to run `SELECT 1 FROM DUAL` using the `dojo/io/script` object. To perfom a HTTP GET request for a JSONP resource the developer creates a `jsonpArgs` object which is passed as an argument to the Dojo function `dojo.io.script.get()`. Dojo takes care of the necessary DOM manipulations to perfom a background request. Upon success, the function `load()`, defined in the object `jsonpArgs` is called. The `jsonpArgs.load()` function is called with the JSON that MySQL sent in reply to `SELECT 1 FROM DUAL`. In case of any errors, dojo calls the `jsonpArgs.error()`. Either callback will display a message on the HTML page by manipulating the contents of the `results` HTML page `div` element.

Please note, for brevity any validation of the data returned from MySQL has been omitted. Also, communication errors are mostly ignored. Production code should take care of possible exceptions:

- MySQL may not return JSON or faulty JSON. Handle JSON parsing errors, if any.
- Valid JSON returned from MySQL is either a result object or an error object. The consumer needs to handle both, not just the result object as in the above example.
- Network errors may occur. Connects may fail, requests may be rejected or time out.

The following script not only includes more debug output and error handling code but also shows how to issue a sequence of SQL commands. Most JavaScript frameworks feature asynchronous background HTTP requests. Should a client sent more than one HTTP request to MySQL at a time, the requests may

be received in arbitrary order by MySQL. They may also be received in the order sent but replies may arrive at the client in arbitrary order.

In many cases, SQL execution order matters. Let there be a sequence such as `DROP TABLE`, `CREATE TABLE`, `INSERT` on the same table. The sequence is supposed to drop a table, recreate it and insert some data into it. Assuming random execution order, it may happen that the `INSERT` statement is executed first. If so, the `INSERT` may either fail because the table it is using does not exist yet or the data inserted into the table will be lost because table is removed in the next step. In any case this is not the desired outcome of the sequence of work. Please note, even if the order of execution is correct, the commands do not execute as a single transaction. There will be no isolation from other concurrently operating clients. All commands are run in autocommit mode.

The example shows a recursive approach to run a sequence of SQL statements ensuring ordering. The list of statements is passed to the function `mysql`. The function takes the first entry from the list, shortens the list by the first entry and issues an asynchronous HTTP request with SQL command from the lists first entry. Only after processing the result of the HTTP request, the function calls itself with the shortened list to process the next SQL command. When processing the results from MySQL, the function performs basic checks of the reply format.

All kinds of network and framework exceptions are handled. Assorted callbacks are registered with the frameworks `notify` functions.

```
<!DOCTYPE html>
<html>
<head>
 <title>Multiple queries</title>
 <meta charset="utf-8">
 <style type="text/css">
 <!--
  body { font-family:sans-serif; font-size:0.8em;  }
  div { padding: 0.5em; margin: 0.5em; }
  .request { font-family:monospace; background-color: #E0E0E0; }
  .reply { font-family:monospace; background-color: #33FF33; }
  .error { font-family:monospace; background-color: #ff3333; }
  .status { background-color: #ffff33; }
 -->
 </style>
</head>
<body>
 <script src="http://ajax.googleapis.com/ajax/libs/dojo/1.10.0/dojo/dojo.js"
  data-dojo-config="async: true"></script>
 <script>
 require(
  ["dojo/dom", "dojo/on", "dojo/request/script", "dojo/request/notify",
  "dojo/json", "dojo/date", "dojo/domReady!"],
  function(dom, on, script, notify, JSON, date) {
   // Results will be displayed in resultDiv
   var resultDiv = dom.byId("resultDiv");
   var statusDiv = dom.byId("statusDiv");
   var startDate, lastDate;

   function mysql(queries) {

    resultDiv.innerHTML += '<div class="request">mysql(';
    if (0 == queries.length) {
     resultDiv.innerHTML += 'No more queries left)<br/>';
     notify_add_diff(resultDiv);
     resultDiv.innerHTML += '</div>';
     return;
    }
```

```
 var query = queries[0];
 queries.shift();
 resultDiv.innerHTML += query + ')<br />';
 resultDiv.innerHTML += queries.length + ' queries left </div>';

 var url = "http://basic_auth_user:basic_auth_passwd@127.0.0.1:8080/sql/myhttp/";
 url += encodeURIComponent(query);
 var promise = script.get(url, {jsonp: "jsonp", query: {jsonp_escape: 1}});
 promise.then(
  function (data) {
   notify_add_diff(resultDiv);
   data = JSON.parse(data, true);

   if (data.server_status) {
    resultDiv.innerHTML += '<div class="reply">';
    resultDiv.innerHTML += "(Status)";
    resultDiv.innerHTML += " Server Status: " + data.server_status;
    resultDiv.innerHTML += " Warnings: " + data.warning_count;
    resultDiv.innerHTML += " Affected Rows: " + data.affected_rows;
    resultDiv.innerHTML += " Last Insert ID: " + data.last_insert_id;
   } else if (data.errno) {
    resultDiv.innerHTML += '<div class="error">';
    resultDiv.innerHTML += "(Error)";
    resultDiv.innerHTML += " Errno: " + data.errno;
    resultDiv.innerHMTL += " SQL State: " + data.sqlstate;
    resultDiv.innerHTML += " Error: " + data.error;
   } else {
    resultDiv.innerHTML += '<div class="reply">';
    if (data.length) {
     resultDiv.innerHTML += "Number of results: " + data.length + "<br />";
     var result_idx;
     for (result_idx = 0; result_idx < data.length; result_idx++) {
      resultDiv.innerHTML += "Result set " + result_idx;
      resultDiv.innerHTML += "<br /><br />";
      resultDiv.innerHTML += JSON.stringify(data[result_idx].data) + "<br /><br />";
      resultDiv.innerHTML += "(Meta) " + JSON.stringify(data[result_idx].meta) + "<br />";
      resultDiv.innerHTML += "(Status) " + JSON.stringify(data[result_idx].status) + "<br />";
      resultDiv.innerHTML += "<hr />";
     }
    } else {
     resultDiv.innerHTML += "(Unknown) ";
     resultDiv.innerHTML += JSON.stringify(data);
    }
   }
   resultDiv.innerHTML += '</div>';
   mysql(queries);
  },
  function (err) {
   resultDiv.innerHTML += '<div class="error">';
   resultDiv.innerHTML += "(Error) " + err;
   resultDiv.innerHTML += '</div>';
   mysql(queries);
  }
 );
}

function notify_add_diff(domDiv) {
 domDiv.innerHTML += date.difference(startDate, new Date(), "millisecond") +
   "ms total, ";
 domDiv.innerHTML += date.difference(lastDate, new Date(), "millisecond") +
   "ms since last<br />";
 lastDate = new Date();
}

notify("error",
 function (error) {
  statusDiv.innerHTML +=  '<div class="error">' + error + '</div>';
```

```
 }
);
notify("start",
 function () {
  statusDiv.innerHTML += '<div class="status">Start ';
  notify_add_diff(statusDiv);
  statusDiv.innerHTML += '</div>';
 }
);
notify("send",
 function (data, cancel) {
  statusDiv.innerHMTL += '<div class="status">Sent request ';
  notify_add_diff(statusDiv);
  statusDiv.innerHTML += '</div>';
 }
);
notify("load",
 function (data) {
  statusDiv.innerHMTL += '<div class="status">Received reply ';
  notify_add_diff(statusDiv);
  statusDiv.innerHTML += '</div>';
 }
);
notify("error",
 function (error) {
  statusDiv.innerHMTL += '<div class="error">Error ';
  statusDiv.innerHTML += error + '<br />';
  notify_add_diff(statusDiv);
  statusDiv.innerHTML += '</div>';
 }
);
notify("done",
 function (data) {
  if (data instanceof Error) {
   statusDiv.innerHMTL += '<div class="error">Done, response processed ';
   statusDiv.innerHMTL += data + '<br />';
  } else {
   statusDiv.innerHMTL += '<div class="status">Done ';
  }
  notify_add_diff(statusDiv);
  statusDiv.innerHTML += '</div>';
 }
);
notify("stop",
 function () {
  statusDiv.innerHMTL += '<div class="status">Stop ';
   notify_add_diff(statusDiv);
   statusDiv.innerHTML += '</div>';
  }
);


// Attach the onclick event handler to the makeRequest button
on(dom.byId('makeRequest'), "click", function (evt) {
 resultDiv.innerHTML = "";
 queries = new Array(
  "DROP TABLE IF EXISTS test",
  "CREATE TABLE test(id INT)",
  "INSERT INTO test(id) VALUES (1), (2), (3)",
  "INSERT INTO test(id) VALUES (4), (5), (6)",
  "INSERT INTO test(id) VALUES (7), (8), (9)",
  "SELECT id FROM test ORDER by id ASC"
 );
 startDate = lastDate = new Date();
 mysql(queries);
 }
);
```

```
 }
);
</script>
<p>Maximum debug info from dojo</p>
<form>
 <input type="button" id="makeRequest" value="Click to query MySQL" />
</form>
<h2>Results</h3>
<div id="resultDiv"></div>
<h2>Status</h2>
<div id="statusDiv"></div>
</body>
</html>
```

Utility code such as result validation and handling should be abstracted and kept in library code that can be reused on occasion. It is likely that such code could be used with most JavaScript frameworks.

Library code plays a vital role in bigger frameworks, such as Dojo. The full potential of Dojo becomes better visible when integrating MySQL in form of a data store library. Assorted Dojo components can then use MySQL as a store. Here, dojo should be considered only an example for many JavaScript frameworks.

The dojo framework provides abstractions for GUI elements such as graphs or spreadsheets. We show two examples for spreadsheet data stores that demonstrate the potential and limits of the HTTP Plugin in this context.

To run the follwing examples download the Dojo Toolkit 1.10 and unpack it. Copy the data store example code from below into the directory `path/to/dojo/dojo-release-1.10.0/store/`. Modify the HTML pages to reference you local copy of the toolkit. Make sure to configure MySQL as described and load the example SQL data. See setup above for details.

The first example shows how to present the rows from a MySQL table in a sortable spreadsheet using the dojo `DataGrid`. The `DataGrid` object takes care of rendering the table and lets you register assorted event handler for user interaction. The data to be displayed is provided by an `ObjectStore`. The `ObjectStore` abstracts the details of the storage used. Data can be stored in a static array, extracted from some file or be kept in a MySQL database. The specifics of the data access are abstracted in library code. This keeps the frontend code comprehensive and allows switching from one data store to another if need be.

```
<!DOCTYPE html>
<html>
<head>
 <title>MySQL dojo store (mapped SQL table)</title>
 <meta charset="utf-8">
 <link rel="stylesheet"
   href="/path/to/dojo/dojo-release-1.10.0/dijit/themes/claro/claro.css">
 <style type="text/css">
  @import "/path/to/dojo/dojo-release-1.10.0/dojox/grid/resources/Grid.css";
  @import "/path/to/dojo/dojo-release-1.10.0/dojox/grid/resources/claroGrid.css";
   #gridDiv {
    height: 10em;
   }
 </style>
</head>
<body class="claro">
 <script src="/path/to/dojo/dojo-release-1.10.0/dojo/dojo.js"
  data-dojo-config="async: true"></script>
 <script>
 require(
  ["dojo/dom-construct", "dojo/parser", "dijit/Dialog",
  "dojox/grid/DataGrid", "dojo/data/ObjectStore",
```

```
  "dijit/form/Select", "dojo/store/Memory", "dojo/dom",
  "dojo/on", "dojo/when", "dojo/request/script",
  "dojo/request/notify", "dojo/json", "dojo/store/JsonpMySQLFields",
  "dojo/domReady!"],
  function(domConstruct, parser, Dialog, DataGrid, ObjectStore,
    Select, Memory, dom, on, when, script, notify, JSON, mysqlp) {

   var store = new mysqlp({
    method: "http",
    host: "127.0.0.1",
    port: 8080,
    interface: "sql/myhttp",
    basicAuthUser: "basic_auth_user",
    basicAuthPassword: "basic_auth_passwd",
    mysqlTable: "dojo_jsonp_fields",
    mysqlFields: ["first_name", "last_name", "email"]
   });

   var ostore = new ObjectStore({ objectStore: store });

   var grid = new DataGrid({
    store: ostore,
    query: {},
    queryOptions: {},
    structure: [
     { name: "First Name", field: "first_name", width: "25%" },
     { name: "Last Name", field: "last_name", width: "25%" },
     { name: "Mail", field: "email", width: "50%" }
    ]}, "gridDiv");
   grid.startup();
  });
 </script>
 <h1>MySQL backed Grid/spreadsheet</h1>
 <form>
  <fieldset>
  <legend>Grid with MySQL contents</legend>
  <div id="gridDiv"></div>
  </fieldset>
 </form>
</body>
</html>
```

The code of the `JsonpMySQLFields` MySQL store used is below. To run the HTML example, create a new file `path/to/dojo/dojo-release-1.10.0/store/JsonpMySQLFields.js` and copy the below into it. Basically, the store maps the dojo internal API for data stores to SQL commands and issues HTTP requests to store and fetch data in MySQL.

```


define("dojo/store/JsonpMySQLFields",
 ["../number", "../_base/array", "../string", "../request/script", "../when",
 "../_base/xhr", "../_base/lang", "../json", "../_base/declare",
 "./util/QueryResults"],
 function(number, array, string, script, when, xhr, lang,
  JSON, declare, QueryResults ) {

  var base = null;

  return declare("dojo.store.JsonpMySQLFields", base, {

  constructor: function(options){
  declare.safeMixin(this, options);
  },

  method: "http",
```

```
host: "127.0.0.1",
port: 8080,
interface: "sql",
basicAuthUser: "basic_auth_user",
basicAuthPassword: "basic_auth_passwd",
mysqlTable: "dojo_jsonp_fields",
mysqlFields: [],
idProperty: "dojo_id",

get: function(oid){
 // SQL INJECTION
 var sql = "SELECT " + this.idProperty + "," + this.mysqlFields.toString();
 sql += " FROM " + this.mysqlTable + " WHERE dojo_id = " + number.parse(oid);
 return when(
  script.get(
   this._getAddress(sql),
    {jsonp: "jsonp"}
  ).then(lang.hitch(this, this._extractFirstRow))
 );
},

getIdentity: function(object){
 return object[this.idProperty];
},

put: function(object, options){
 options = options || {};

 var sql = "";
 var values = "";
 var id = ("id" in options) ? options.id : this.getIdentity(object);
 var hasId = typeof id != "undefined";

 if (("overwrite" in options) && options["overwrite"]) {
  if (!hasId) {
   throw "You must provide the id of the object to update";
  }

  array.forEach(this.mysqlFields, lang.hitch(this, function (field) {
   if (field in object) {
    values += field + "=";
    values += "'" + this._escapeString(object[field]) + "', ";
   }
  }));
  if (values.length == 0) {
   throw "Object has no known property for SQL column mapping";
  }

  sql = "UPDATE " + this.mysqlTable + " SET " + values + " version = version + 1";
  sql += " WHERE " + this.idProperty + "= " + number.parse(id);

} else {
 var fields = "";

 if (hasId) {
  fields += this.idProperty + ", ";
  values += number.parse(id) + ", "
 }

 array.forEach(this.mysqlFields, lang.hitch(this, function (field) {
  if (field in object) {
   fields += field + ", ";
   values += "'" + this._escapeString(object[field]) + "', "
  }
 }));
 if (fields.length == 0) {
  throw "Object has no known property for SQL column mapping";
```

```
 }

 sql = "INSERT INTO " + this.mysqlTable + "(";
 sql += fields.substring(0, fields.length - 2);
 sql += ") VALUES (" + values.substring(0, values.length - 2) + ")";
}
return when(
 script.get(
  this._getAddress(sql),
  {jsonp: "jsonp" }
 ).then(
   function (reply) {
    if (reply && "last_insert_id" in reply)
     return reply.last_insert_id;
    return reply;
   }
  )
 );
},

add: function(object, options){
 options = options || {};
 options.overwrite = false;
 return this.put(object, options);
},

remove: function(id, options){
 options = options || {};
 var sql = "DELETE FROM " + this.mysqlTable + " WHERE " + this.idProperty + "=" + number.parse(id);
 return when(
  script.get(
   this._getAddress(sql),
   {jsonp: "jsonp" }
  ).then(
    function (reply) {
     if (reply && "errno" in reply) {
      return reply;
     }
     return;
    }
  )
 );
},

query: function(query, options){
 options = options || {};
 var sql = "SELECT " + this.mysqlFields.toString() + ", " + this.idProperty;
 sql += " FROM " + this.mysqlTable;

 if (options) {
  if (options.sort && options.sort.length) {
   var order_by = "";
   for (var i = 0; i< options.sort.length; i++) {
    if (order_by.length)
     order_by += ", ";
    var sort = options.sort[i];
    order_by += sort.attribute;
    order_by += (sort.descending) ? " DESC" : " ASC";
   }
   if (order_by.length)
    sql += " ORDER BY " + order_by;
  }

  if (options.start >= 0 || options.cout >= 0) {
   var limit = "";
   if (options.start)
    limit += options.start;
```

```
    if ("count" in options && options.count != Infinity) {
     if (limit.length)
      limit += ", ";
     limit += options.count;
    }
    if (limit.length)
     sql += " LIMIT " + limit;
   }
  }
  var results = when(
   script.get(
    this._getAddress(sql),
    {jsonp: "jsonp" }
   ).then(lang.hitch(this, this._extractAllRows)));
  return QueryResults(results);
 },

 _getAddress : function(query) {
  return this.method + "://" + this.basicAuthUser + ":" +
    this.basicAuthPassword + "@" + this.host + ":" + this.port +
    "/" + this.interface + "/" + encodeURIComponent(query);
 },

 _extractRows : function(result, limit) {
  var data_only = new Array();
  var result_idx, row_idx;
  var object;

  if (result && "errno" in result) {
   data_only.push(result);
   return data_only;
  }

  for (result_idx = 0; result_idx < result.length; result_idx++) {
   if ("errno" in result[result_idx]) {
    data_only.push(result[result_idx]);
   } else {
    for (row_idx = 0; row_idx < result[result_idx].data.length; row_idx++) {
     if ((limit > 0) && (row_idx >= limit)) {
      return data_only;
     }
     tmp = new Object;
     array.forEach(result[result_idx].data[row_idx], function (value, column_idx) {
      tmp[result[result_idx].meta[column_idx].column] = value;
     });
     data_only.push(tmp);
    }
   }
  }
  return data_only;
 },

 _extractAllRows: function (result) {
  return this._extractRows(result, -1);
 },

 _extractFirstRow: function (result) {
  return this._extractRows(result, 1);
 },

 _escapeString: function (sql_value) {
  sql_value = sql_value.toString();
  return sql_value.replace('/"/g', '\\"');
 }
});

});
```

The sketched store has two limitations. For simplicity, possible SQL injection risks are not taken care of. The `_escapeString` method will not catch all possible risks. Additional input filtering should be done. While it is only a matter of time to secure it, handling totally unstructured data that requires extreme schema flexibility is more difficult.

Instead of mapping the columns of one or more relational tables to the columns of a spreadsheet one can also store all data in an unstructured JSON document inside a BLOB column of a relational table. Keeping JSON documents inside a BLOB fulfills the criteria of extreme schema flexibility but MySQL offers no SQL expressions to search the JSON documents. However, with a key-value centric API like that of a dojo store and the option to do the search at runtime in the store itself, it may be still an option. Below is a variation of the previous store. The below can be used to store arbitrary JSON in a BLOB column.

Note that this approach is similar to what the JSON Document (DOC) endpoint offers.

```
define("dojo/store/JsonpMySQL",
 ["../number", "../string", "../request/script", "../when", "../_base/xhr",
 "../_base/lang", "../json", "../_base/declare",
 "./util/QueryResults"],
function(number, string, script, when, xhr, lang, JSON, declare, QueryResults ){

 var base = null;
 return declare("dojo.store.JsonpMySQL", base, {

  constructor: function(options){
   declare.safeMixin(this, options);
  },

  method: "http",
  host: "127.0.0.1",
  port: 8080,
  interface: "myhttp",
  basicAuthUser: "basic_auth_user",
  basicAuthPassword: "basic_auth_passwd",
  mysqlTable: "dojo_jsonp",
  mysqlBlob: "dojo_blob",
  mysqlId: "dojo_id",
  target: "",
  idProperty: "id",

  get: function(oid){
   var sql = "SELECT " + this.mysqlId + ", " + this.mysqlBlob + "FROM";
   sql += this.mysqlTable + " WHERE " + this.mysqlId + "=" + number.parse(oid);

   return when(
    script.get(
     this._getAddress(sql),
     {jsonp: "jsonp"}
    ).then(lang.hitch(this, this._extractFirstObject))
   );
  },

  getIdentity: function(object){
   return object[this.idProperty];
  },

  put: function(object, options){
   options = options || {};
   var sql = "";
   var id = ("id" in options) ? options.id : this.getIdentity(object);
   var hasId = typeof id != "undefined";
```

```
 if (("overwrite" in options) && options["overwrite"]) {
  if (!hasId) {
   throw "You must provide the id of the object to update";
  }

  sql  = "UPDATE " + this.mysqlTable + " SET " + this.mysqlBlob + "= '";
  sql += this._escapeString(JSON.stringify(object)) + "'";
  sql += " WHERE " + this.mysqlId + "=" + number.parse(id);

 } else {
  sql = "INSERT INTO " + this.mysqlTable + "(" + this.mysqlBlob;
  if (hasId) {
   sql += ", " + this.mysqlId;
  }
  sql += ") VALUES ('" + this._escapeString(JSON.stringify(object)) + "'";
  if (hasId) {
   sql += ", " + number.parse(id);
  }
   sql += ")";
 }

 return when(
  script.get(
   this._getAddress(sql),
   {jsonp: "jsonp" }
  ).then(
   function (reply) {
    if (reply && "last_insert_id" in reply)
     return reply.last_insert_id;
    return reply;
   }
  )
 );
},

add: function(object, options){
 options = options || {};
 options.overwrite = false;
 return this.put(object, options);
},

remove: function(id, options){
 options = options || {};
 var sql = "DELETE FROM " + this.mysqlTable + " WHERE " +
   this.mysqlId + "=" + number.parse(id);
 return when(
  script.get(
   this._getAddress(sql),
   {jsonp: "jsonp" }
   ).then(
    function (reply) {
     if (reply && "errno" in reply) {
      return reply;
     }
     return;
    }
  )
 );
},

query: function(query, options){
 var sql = "SELECT " + this.mysqlId + ", " + this.mysqlBlob;
 sql += " FROM " + this.mysqlTable;

 options = options || {};

 if (options) {
```

```
    if (options.start >= 0 || options.cout >= 0) {
     var limit = "";
     if (options.start)
      limit += options.start;
     if ("count" in options && options.count != Infinity) {
      if (limit.length)
       limit += ", ";
      limit += options.count;
     }
     sql += " LIMIT " + limit;
    }
   }
   var results = when(
    script.get(
     this._getAddress(sql),
     {jsonp: "jsonp" }
    ).then(lang.hitch(this, this._extractAllObjects)));

   return QueryResults(results);
  },

  _getAddress : function(query) {
   return this.method + "://" + this.basicAuthUser + ":" +
     this.basicAuthPassword + "@" + this.host + ":" + this.port +
     "/" + this.interface + "/" + encodeURIComponent(query);
  },

  _extractObjects : function(result, limit) {
   var data_only = new Array();
   var result_idx, row_idx;

   for (result_idx = 0; result_idx < result.length; result_idx++) {
    for (row_idx = 0; row_idx < result[result_idx].data.length; row_idx++) {
     if ((limit > 0) && (row_idx >= limit)) {
      return data_only;
     }
     data_only.push(JSON.parse(result[result_idx].data[row_idx][1]));
    }
   }
   return data_only;
  },

  _extractAllObjects : function (result) {
   return this._extractObjects(result, -1);
  },
  _extractFirstObject: function (result) {
   return this._extractObjects(result, 1);
  },

  _escapeString: function (sql_value) {
   sql_value = sql_value.toString();
   return sql_value.replace('/"/g', '\\"');
  }
 });
});
```

To use it, save it under `path/to/dojo/dojo-release-1.10.0/store/JsonpMySQL.js`. Modify the HTML spreadsheet example to load the store instead of the previous `JsonpMySQLFields`. Replace the store used by the spreadsheet. Note that this version of the spreadsheet is not sortable.

```
var store = new mysqlp({
 method: "http",
 host: "127.0.0.1",
```

```
 port: 8080,
 interface: "sql/myhttp",
 basicAuthUser: "basic_auth_user",
 basicAuthPassword: "basic_auth_passwd",
 mysqlTable: "dojo_jsonp",
});
```

4.3.6 Limitations and pitfalls

The following differences, limitations and pitfalls have been identified by comparing the behaviour of the HTTP Plugin with that of the MySQL C API.

When using the MySQL C API the SQL function `CONNECTION_ID()` returns the id of the MySQL internal SQL thread processing the SQL function itself. The return value can be used, for example, to kill the current connection or it issue question such as `SELECT COMMAND, STATE, INFO FROM INFORMATION_SCHEMA.PROCESSLIST WHERE ID = CONNECTION_ID()`. Any such query will not work properly when using the HTTP Plugin. With the HTTP Plugin `CONNECTION_ID()` is not guaranteed to return the correct internal thread id. This is due to the way the prototype is implemented. There is no known workaround.

Wrong results can be observed with binary data. The prototype tries to convert binary data on a per character basis into UTF-8. If the conversion fails, it silently skipps the character. Thus binary results be converted into empty or in random other strings that omit input data. This is a serious issue. Conversion works flawless if the MySQL server returns numbers to the plugin using a binary charset (more precisely: charset 63). This is, for example, the case with `LAST_INSERT_ID()`. All SQL functions that return binary strings are likely to show wrong results when called through the HTTP Plugin. This includes `COMPRESS()`, `DECODE()`, `INET6_ATON()`. Other functions may be affected as well. If in doubt, users should check the meta data of a column in question to decide whether the result could be flawed. This issue is being worked on.

4.4 The CRUD endpoint: /crud/

The CRUD endpoint allows basic data manipulation of records from tables that have a non-compound primary key defined. Table records can be created, read, updated and deleted. Records are identified by their primary key. Only tables that define a single column primary keys can be access through the endpoint. `NULL` or empty strings are not allowed as primary key values.

HTTP methods are mapped to corresponding data manipulation operations.

- A HTTP `PUT` request either creates new records or updates existing ones.
- To read records a `GET` request is used.
- Records are removed using `DELETE` HTTP requests.

The interface cannot be used to perfrom data definition language (DDL) operations. That means, you cannot, for example, send administrative commands, create schema object or modify user accounts. Only data manipulation is allowed.

Valid requests result in replies with either no content or carry a valid JSON document. The only charset supported is UTF-8, all input data should use UTF-8 and all replies will use UTF-8. At the time of writing, HTTP requests are mapped to SQL statements internally. All user commands follow autocommit logic, because HTTP is stateless and offers no concept of a session.

Query results do not contain any meta data information but data only to reduce their length.

All endpoints use the same security and user concept. HTTP basic authentication must be used with all connections. Any successfully authenticated non-SSL connection can cause MySQL to carry out actions as a predefined MySQL user. See above for SSL notes and further details.

4.4.1 API overview

The CRUD endpoint listens to HTTP(S) GET, PUT and DELETE commands on the host and port of the HTTP Plugin, if the request URL prefix does match the server variable `myhttp_curl_url_prefix`. Assuming default settings are used for plugins server variables, the CRUD endpoint answers to all URLs that begin with `http://127.0.0.1:8080/crud/` and `http://127.0.0.1:8080/crud/`. The full URL pattern for requests is `protocol://host:port/crud/database/table/` optionally followed by the primary key of the record to be modified.

All endpoint expect a default database to be set as part of the URL: `protocol://host:port/crud/database/`. The database URL element is mandatory but it can be an empty string in which case, the default database is `myhttp_default_db`.

A primary key must be given as part of the URL for all GET, PUT and DELETE requests. The URL pattern for any such request is `protocol://host:port/crud/database/table/pk`. Below is an example how to use PUT to create a record. The record will be added to the table `simple` from the database `myhttp` and be stored under the primary key `101`. Please, note that a PUT request replaces an already existing record with the same primary key value. Furthermore the return code for a successful PUT is always `200 OK` although `201 Created` might be a better choice if the record has been newly added by the HTTP request. Details may change in future versions.

```
shell> curl -H "Accept: application/json" -X PUT -d '{ "col_a": "Yet another example"}'
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/simple/101'

{
  "affected_rows": 1,
  "warning_count": 0
}
```

To read the newly created record, follow the URL pattern `protocol://host:port/crud/database/table/pk` and issue a GET request for `http://127.0.0.1:8080/crud/myhttp/simple/101`

```
shell>  curl --user basic_auth_user:basic_auth_passwd
   --url "http://127.0.0.1:8080/crud/myhttp/simple/101"

{
  "id": "101",
  "col_a": "Yet another example"
}
```

The CRUD endpoint does not return any meta data with the record but only the data of the record itself. Therefore the size of the reply will be smaller than that from the SQL and Document endpoints. Network traffic is less.

The newly created record can be removed by sending a DELETE HTTP request.

```
shell> curl -v -X DELETE
  --user basic_auth_user:basic_auth_passwd
```

```
  --url 'http://127.0.0.1:8080/crud/myhttp/simple/101'
[...]

* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> DELETE /crud/myhttp/simple/101 HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 0
< Pragma: no-cache
< Content-Type: application/json
```

The following table gives SQL equivalents for the CRUD endpoint actions. The HTTP Plugin for MySQL maps CRUD endpoint requests to the SQL commands shown. Performance optimization of the internal implementation are possible. But, the simple mapping to SQL allows setting the focus on the user API and to adapt it very quickly at this early development stage.

CRUD endpoint	SQL equivalent
Create (and replace): PUT	`REPLACE INTO db.table SET ..., pk = ...`
Read: GET	`SELECT * FROM db.table WHERE pk = ...`
Delete: DELETE	`DELETE FROM db.table WHERE pk = ...`

4.4.2 HTTP methods, headers and status codes

The following HTTP headers are set by the CRUD endpoint. The headers do not differ from those set by the SQL endpoint.

Header	Description
Server	Always given. Can be considered as an API version. For example: `MyHTTP 1.0.0-alpha`
Cache-control	Always given. Always `must-revalidate`
Pragma	Always given. Always `no-cache`
Content-Length	Always given.
Content-Type	Set to `application/json` for `200 OK`, `400 Bad Request`, `401 Unauthorized`. Other replies may or may not contain it.
Connection	Always given. Always `Keep-Alive`.

The CRUD endpoint supports GET, PUT and DELETE requests. Other requests will be rejected with code `405 Method Not Allowed`. The reply to such a rejected request contains no content. No error message is given.

Resources are protected with HTTP basic authentication. Should HTTP basic authentication fail or not be used with the request at all, the server replies with code `401 Unauthorized` and the JSON error message `{"errno":1045, "sqlstate":"28000","error":"401 Unauthorized"}`.

A request that uses an allowed method but has a malformed URL results in a code `400 Bad Request` reply. The GET, DELETE and PUT methods must be used with an URL following the pattern `protocol://host:port/crud/database/table/primary_key_value`. The code is also used to indicate a general error. An error reply may show no contents or show a JSON error message document that gives details.

PUT methods must sent a JSON object to the server. The JSON object must be flat, which means its members must use scalar datatypes. The members must either be of type string or null. For example, if you want to insert a number in a column of the SQL type `FLOAT` you cannot provide a JSON number:

```
shell> curl -v -d '{"col_float": 0.123}'
  -X PUT --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/sql_types/101'

[...]
< HTTP/1.1 400 Bad Request
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 0
< Pragma: no-cache
< Content-Type: application/json
<
```

Instead, the number must be given as a string. Please, expect this limitation to be lifted soon.

```
shell> curl -v -d '{"col_float": "0.123", "col_char": "a",
  "col_date" : "2015-09-16 15:18:32", "col_decimal": "1.23",
  "col_bigint": "12345678"}'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/sql_types/101'

* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> PUT /crud/myhttp/sql_types/101 HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
> Content-Length: 124
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 124 out of 124 bytes
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 37
< Pragma: no-cache
< Content-Type: application/json
<
{"affected_rows":1,"warning_count":1}
```

The names of the input object members must match the column names of the underlying table. If a table has two columns `col_a`, `col_b` in addition to a primary key column `id`, then the JSON input object may have at most two members: `{"col_a": ..., "col_b": ...}`. Adding a member with the name of the

primary key column is an error. The primary key value is always given as part of the URL but never as an object member.

```
shell> curl -X PUT -d '{"id":"1","col_a":"Hello"}'
 --user basic_auth_user:basic_auth_passwd
 --url 'http://127.0.0.1:8080/crud/myhttp/simple/1'

{
  "errno": 1110,
  "error": "Column 'id' specified twice"
}
```

The input JSON object of a PUT command must include members and values for all columns, but the primary key column, from the underlying that have no default value. For example, the example table `simple` has two columns. The primary key column `id` and column `col_a`. Thus, a valid JSON input object may have at most one member `col_a`. However, the member is not mandatory because a default value has been defined for the underlying SQL column. An empty JSON object can be used as input.

```
mysql> SHOW CREATE TABLE simple\G

*************************** 1. row ***************************
Table: simple
Create Table: CREATE TABLE `simple` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `col_a` varchar(255) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8
```

The use of `AUTO_INCREMENT` primary key columns is not supported. Tables may have a primrary column defined with the column flag set but cients must always provide the primary key value in the URL. The CRUD endpoint currently cannot be used in such a way that an auto increment column generates a primary key value on insert.

The table summarizes the supported HTTP request methods:

HTTP request method	Description
GET	Use standard URL pattern. No input data, no special request headers.
PUT	Use standard URL pattern. No special request headers. Input data must be valid JSON: • Input must be a flat JSON object: object members must be string or null. • For every column of the underlying that has no default value defined, there must be a member of the same name as the column. • Primary key must not be part of the object.
DELETE	Use standard URL pattern. No input data, no special request headers.

4.4.3 JSON with padding (JSONP)

The CRUD endpoint supports JSON with padding for GET requests. Other requests ignore the JSONP URL parameters. The URL parameters are the same as for the SQL endpoint:

Parameter	Description
jsonp=<callback>	Enables JSON with padding. The parameters value is used as the callback function name. Replies returned from the plugin change from `reply` to `callback(reply)`.
jsonp_escape	Only considered when `jsonp` is also given. Replies returned from the plugin change from `reply` to `callback("reply")` with `reply` being escaped appropriately.

4.4.4 JSON content formats

Any valid request is supposed to return a valid JSON document. A valid request is one that uses one of the supported HTTP methods and a well formed URL for the HTTP method. A valid JSON document is defined as a JSON array or object. Please, see the SQL endpoint description for further details on the JSON standard.

There are three types of JSON documents: result documents, error documents and status documents.

```
REPLY:
 jsonp_function(answer) |
 answer

jsonp_function:
  string

answer:
  resultset |
  error |
  status
```

The structure of the documents is extremly simple. The documents can be very small in size due to their simplicity and the omission of meta data. JSON result documents returned by the CRUD endpoint will always be smaller than those returned from the SQL endpoint. Unless JSON unicode encoding causes overhead, replies will also be smaller than standard replies to clients using the MySQL Client/Server Protocol. The lower network traffic may bare some performance advantage. However, please note, that the CRUD endpoints result document always contains a complete row whereas a SQL user may fetch selected columns only. Also, performance has not been a goal in the development of the HTTP Plugin.

Clients can often check the HTTP return code to determine the structure of the HTTP return contents, if any. For example, a `405 Method Not Allowed` reply is not expected to be accomplished with any JSON content. Correct requests will often return `200 OK` together with a JSON result, error or status document. The request method helps to further narrow the list of valid replies in case of `200 OK`. Only a GET request is expected to return a result. A successful PUT request returns a status document, whereas a successful DELETE returns no contents.

HTTP return code	Content	Methods	Details
401 Unauthorized	Error document or no content	All	Wrong HTTP basic authentication credentials
400 Bad Request	Error document	All	Check error message.
405 Method Not Allowed	No content	Any but GET, PUT, DELETE	
200 OK	Result document	GET	
200 OK	Status document	PUT	
200 OK	No content	DELETE	

HTTP return code	Content	Methods	Details
404 Not Found	Error document	GET, DELETE	

4.4.5 JSON result document

The JSON result document is the JSON equivalent of a single table row. It contains only the row data, there is no meta data. The result document consists of one object member for each column in the row. The member name equals the column name.

```
resultset:
{
  "column_name" : column_value (, "column_name" : column_value ...)
}

column_name:  string

column_value:
  string |
  null
```

Column values are returned as strings or null. Underlying SQL column data types are not mapped to JSON data types. Because of the omission of meta data for simplicity and reduction of network traffic, clients cannot automatically map the underlying SQL column types into data types of the target domain. If you need meta data, consider using the SQL endpoint instead.

```
shell>  curl --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/crud/myhttp/sql_types/1"

{
  "id": "1",
  "col_char": "CHAR(127)",
  "col_null": null,
  "col_date": "2014-08-21",
  "col_decimal": "123.45",
  "col_float": "0.9999",
  "col_bigint": "9223372036854775807"
}
```

JSON result documents are produced by GET requests only.

4.4.6 JSON error document

The CRUD endpoint may reply with a JSON error document to failed requests. Error documents should be expected when the return code is `400 Bad Request` or `401 Unauthorized`. The error document is a JSON object with two members: `errno`, `error`.

```
error:
{
 "errno": int,
 "error": string
}
```

The error number `errno` and error description `error` often show MySQL server error codes and messages. The CRUD endpoint is using SQL commands to execute some HTTP requests. Should a SQL command fail its error code and message are reported in the HTTP reply.

```
shell> curl -v  --user basic_auth_user:basic_auth_passwd
  --url "http://127.0.0.1:8080/crud/test/test/1"

[...]
< HTTP/1.1 400 Bad Request
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 94
< Pragma: no-cache
< Content-Type: application/json
<
{
  "errno": 1044,
  "error": "Access denied for user 'http_sql_user'@'127.0.0.1' to database 'test'"
}
```

In some cases the MySQL server has no adequate error code defined. Then, the HTTP Plugin will use a generic error code such as `2000` and provide it's own error message. A malformed HTTP request URL is an example of an error condition for which the server has no standard error code. This is an exception that does not arise with MySQL normally.

```
shell> curl --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/simple/'

{
  "errno": 2000,
  "error": "The request URL must include a primary key value"
}
```

4.4.7 JSON status document

A successful HTTP PUT request for the CRUD endpoint returns code `200 OK` together with a JSON status document. The status object has two members. The `affected_rows` member reports the number of modified rows. The `warning_count` is the number of SQL warnings caused by the SQL statement that the CRUD endpoint used internally to perform the requested action. Both values are reported by the server. Please, consult the documentation of the MySQL C API functions `mysql_affected_rows()` and `mysql_warning_count()` for more details.

```
error:
{
 "affected_rows": int,
 "warning_count": int
}
```

Below is an example of a status document that has been sent in reply to adding a record to a table from the example setup. The PUT command affects one row in the table `simple` from the database `myhttp`. There was neither a SQL error not a SQL warning during the execution.

```
shell> curl -X PUT -d '{"col_a":"Hello"}'
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/simple/1'

{
  "affected_rows": 1,
```

```
  "warning_count": 0
}
```

4.4.8 Commandline examples

The CRUD endpoint offers key - row access semantics. All data is access through a primary key. You cannot search for data in any other way but by key. All data is stored in relational tables. An access to a row always returns all columns. Returned rows include no meta data, data type information is lost.

The properties of the endpoint can be read as a limitation or understood as a feature. The CRUD endpoint qualifies for applications that deal with highly structured data. It is not possible for a client to burden the database server with complex queries. Only fast primary key based operations are permitted.

Data definition statements or general administrative statements cannot be executed, which further restricts the use cases. The absence can be seen as an additional security feature. Functionality is not made available to the user because of the limited API. Accesses through the API are additionally secured using standard user account management features.

To get started with the CRUD interface, create a table `crud_messages` in the database `myhttp`, which is the example database used for the HTTP Plugin. Setup instructions for the example database and required plugin settings are given above. Do not define a primary key for the table `crud_messages`. Insert a record.

```
mysql> CREATE TABLE `crud_messages` (
  `message_id` int(11) NOT NULL,
  `created` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `sender` varchar(127) NOT NULL DEFAULT 'anonymous',
  `message` varchar(1024) DEFAULT NULL
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
mysql> INSERT INTO crud_messages(message_id, message) VALUES (1, 'Three-wheelers are fun');
```

The CRUD endpoint URL pattern for accessing a table is `http//server:port/crud/database/table/pk` which means, a HTTP GET access for `http//127.0.0.1:8080/crud/myhttp/crud_messages/1` is the equivalent to `SELECT * FROM myhttp.crud_messages WHERE pk_col = 1` with `pk_col` being the name of the tables primary key column. Because no primary key has been defined for the table `crud_messages` the HTTP request fails with code `400 Bad Request` and an error document that gives details.

```
shell> curl -v --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/crud_messages/1'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> GET /crud/myhttp/crud_messages/1 HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 400 Bad Request
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 61
< Pragma: no-cache
< Content-Type: application/json
<
```

```
{
  "errno": 1173,
  "error": "This resource requires a primary key"
}
```

Add a primary key to the table `myhttp`.

```
mysql> ALTER TABLE crud_messages ADD PRIMARY KEY(message_id);
```

Retry the HTTP request. It will now return the row identified by the primary key value `1`.

```
mysql> curl -v --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/crud_messages/1

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> GET /crud/myhttp/crud_messages/1 HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 107
< Pragma: no-cache
< Content-Type: application/json
<
{
  "message_id": "1",
  "created": "2014-08-28 19:37:58",
  "sender": "anonymous",
  "message": "Three-wheelers are fun"
}
```

The HTTP PUT method can be used to add records. Below is the equivalent of `REPLACE INTO myhttp.crud_messages SET message_id = 2, sender = 'Ulf', message = 'My three-wheeler is broken :-('`. Please note, the HTTP request does not contain input data for all columns. The primary key column value is always derived from the URL `http://127.0.0.1:8080/crud/myhttp/crud_messages/2`: `message_id = 2`. The values for `sender` and `message` are taken from the JSON input `{"sender": "Ulf", "message": "My three-wheeler is broken :-("}`. It is not mandatory to include a value for the `created` column, because the columns SQL definition `created` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP` includes a default.

In reply to the command the server sends `200 OK` and a status message which confirms that one row was affected.

```
shell> curl -v -X PUT
  -d '{"sender": "Ulf", "message": "My three-wheeler is broken :-("}'
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/crud_messages/2'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
```

```
* Server auth using Basic with user 'basic_auth_user'
> PUT /crud/myhttp/crud_messages/2 HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
> Content-Length: 62
> Content-Type: application/x-www-form-urlencoded
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 37
< Pragma: no-cache
< Content-Type: application/json
<
{
  "affected_rows": 1,
  "warning_count": 0
}

shell> curl --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/crud_messages/2'

{
  "message_id": "2",
  "created": "2014-08-29 10:07:05",
  "sender": "Ulf",
  "message": "My three-wheeler is broken :-("
}
```

A new PUT request for the same URL can be issued to replace the record stored under the URL. There is no warning that existing data will be replaced. The value of 2 reported for `affected_rows` hints that a record existed before and has been replaced. Two rows are affected: the old record and the newly inserted one. The `affected_rows` value is provided by the server.

```
shell> curl -X PUT -d '{"sender": "Andrey", "message":"Hmm"}'
 --user basic_auth_user:basic_auth_passwd
 --url 'http://127.0.0.1:8080/crud/myhttp/crud_messages/2'

{
  "affected_rows": 2,
  "warning_count": 0
}

shell> curl --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/crud_messages/2'

{
  "message_id": "2",
  "created": "2014-08-29 10:09:36",
  "sender": "Andrey",
  "message": "Hmm"
}
```

The PUT method cannot be used to update a record partially. Andrey's message cannot be changed with a JSON object that is made of a `message` member only. This will create a new record with the servers defaults for `sender` and `created`, the `message` set to `Enjoy your weekend` and the primary key value of `2`

```
shell> curl -X PUT -d '{"message":"Enjoy your weekend"}'
 --user basic_auth_user:basic_auth_passwd
 --url 'http://127.0.0.1:8080/crud/myhttp/crud_messages/2'

{
  "affected_rows": 2,
  "warning_count": 0
}

shell> curl --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/crud_messages/2'

{
  "message_id": "2",
  "created": "2014-08-29 11:11:51",
  "sender": "anonymous",
  "message": "Enjoy your weekend"
}
```

Use DELETE requests to remove records. The return code indicates whether a record has been removed or you tried to access a non-existing record.

```
shell> curl-v -X DELETE
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/crud_messages/2'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> DELETE /crud/myhttp/crud_messages/2 HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 0
< Pragma: no-cache
< Content-Type: application/json

shell> curl -v -X DELETE
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/crud/myhttp/crud_messages/2'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> DELETE /crud/myhttp/crud_messages/2 HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Connection: Keep-Alive
< Cache-control: must-revalidate
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 0
< Pragma: no-cache
```

4.4.9 JavaScript examples: basics, AngularJS

The Lab release of the HTTP Plugin does not support the CORS standard to overcome the same origin policy. Most JavaScript programs will have to use JSONP to access the CRUD endpoint. The CRUD endpoint supports JSONP only with GET requests. This restricts JavaScript clients to read requests in this early version.

4.4.9.1 Reading records with AngularJS and JSONP

The AngularJS JavaScript framework describes itself as a structural framework for dynamic web applications. The static declarative nature of HTML is overcome by extending HTML. HTML becomes a template language. According to the authors of the framework, the framework was built with CRUD applications in mind.

The APIs of the HTTP Plugin for MySQL are anything but finalized. More features, such support for CORS, are required to create CRUD applications using MySQL and AngularJS. Still, it is already possible to query MySQL from AngularJS.

The below example shows how to use the AngularJS $http service to fetch a row identified by the primary key value of `1` from the MySQL table `simple` of the `myhttp` database through the CRUD endpoint. AngularJS applications encourage the use of the Model-View-Controller (MVC) design pattern. In AngularJS terms, the HTML page of the example is a template. The view is the DOM manipulated version of the template, the model is the data shown to the user and the controller encapulated the business logic on the model.

```
<!DOCTYPE html>
<html ng-app="" ng-controller="crud">
<head>
    <title>AngularJS CRUD example </title>
    <meta charset="utf-8">
</head>
<body>
  <script language="JavaScript" src="http://code.angularjs.org/1.2.9/angular.min.js"></script>
  <script language="JavaScript">

    // Utility to generate a CRUD endpoint request URL
    function get_mysql_url(table, key) {
      return "http://basic_auth_user:basic_auth_passwd@" +
        "127.0.0.1:8080/crud/myhttp/" +
        table + "/" + key + "?jsonp=JSON_CALLBACK";
    }

    // "Main" (controller)
    function crud($scope, $http) {
      var url = get_mysql_url("simple", 1);
      $http.jsonp(url)
        .success(
          function (data, status, headers, config) {
            if (data && data.col_a) {
              $scope.reply = data.col_a;
            } else {
              $scope.reply = "Unknown reply format :-(";
            }
          })
        .error(
          function (data, status, headers, config) {
            if (data && data.error) {
              $scope.reply = "Error: " + data.error;
            } else {
              $scope.reply = "Unkown error";
            }
```

```
        }
      );
  }
</script>
<p>
  {{ reply }}
</p>
</body>
</html>
```

The example uses the entire HTML page as a template for the view. The views controller is set with `<html ng-app="" ng-controller="crud">`. Strictly speaking, `crud` is a factory function that creates an instance of a controller assigned to the view.

The `crud` controller does nothing but use the built-in `$http` service to perform a JSONP HTTP request. The JSONP request is performed through the shortcut method `jsonp`, which takes an URL and an optional config object as its parameters. The `jsonp` method returns a HTTP Promise. Promises are a standard technique for handling asynchronous operations. The HTTP Promise has two HTTP specific methods `success` and `error`. The `success` method will be called when MySQL returns a status code between 200 and 299, as it is the case for a successful CRUD endpoint read. When called, `success` checks the data it gets from MySQL and sets the `reply` property of scope. For our purposes, the scope can be considered glue code between the model and the view. Here, we use the scope in a one way fashion to exchange data between the controller and the view. Whenever the scope property is modified, AngularJS renders the property value into the bound `{{ reply }}` expression.

The next example discusses a standard pitfall of asynchronous APIs such as HTTP APIs. Assume you want to read all rows from the table `simple` ordered by the primary key column `id`.

```
mysql> SELECT * FROM simple ORDER BY id ASC;

+----+--------+
| id | col_a  |
+----+--------+
|  1 | Hello  |
|  2 |        |
|  3 | world! |
+----+--------+
```

Three HTTP requests need to be issued to read the rows through the CRUD interface. To receive an ordered list of rows it is not sufficient to sent the requests in primary key column order. MySQL may receive the requests in any order and reply in any order. Results can be shuffled. A client can overcome this problem in several ways. For example, the client can wait for all results to arrive and reorder them. Or, the client can convert the asynchronous execution into a synchronous one. The use of promise chaining to turn an asynchronous execution into a synchronous one is demonstrated in the SQL endpoint chapter using the Dojo framework.

Therefore, the below example uses the sort approach. As before, Angulars built-in `$http` service is used to query the HTTP Plugin CRUD endpoint. All results received are stored in an array. The array is indexed by the primary key value and the view displays array values in index order.

```
<!DOCTYPE html>
<html ng-app="" ng-controller="crud">
<head>
    <title>AngularJS CRUD example </title>
```

```
    <meta charset="utf-8">
</head>
<body>
  <script language="JavaScript" src="http://code.angularjs.org/1.2.9/angular.js"></script>
  <script language="JavaScript">

    // Utility to generate a CRUD endpoint request URL
    function get_mysql_url(table, key) {
      return "http://basic_auth_user:basic_auth_passwd@" +
        "127.0.0.1:8080/crud/myhttp/" +
        table + "/" + key + "?jsonp=JSON_CALLBACK";
    }

    // Create a HTTP Promise for accessing MySQL
    function get_mysql_crud_http_promise(http, table, key, replies) {
      var url = get_mysql_url(table, key);
      return http.jsonp(url, {id: key, result: replies})
        .success(
          function (data, status, headers, config) {
            // Transform the MySQL reply and add to results list
            config.result[config.id] = {
              key: config.id,
              msg: data.col_a
            };
        })
        .error(
          function(data, status, headers, config) {
            // HTTP reply code is not 2xx - we may or may not have data
            config.result[config.id] = {
              key: config.id,
              msg: "Error: " + ((data && data.error) ? data.error : "Unknown error")
            };
          }
        )
    }


    // "Main" (controller)
    function crud($scope, $http) {
      var replies = new Array(4);
      var key;

      replies[0] = {msg: "Fetching records from table simple...", key: "n/a"};
      $scope.greetings = replies;

      // Fetch all three records from the table simple
      for (key = 1; key <= 3; key++) {
        get_mysql_crud_http_promise($http, "simple", key, replies)
          .then($scope.greetings = replies);
      }
    }
  </script>
  <p>
  <ul>
    <li ng-repeat="msg in greetings track by $index">
      {{ msg.msg }}<br /><span style="font-size:50%">(key: {{ msg.key }})</span>
    </li>
  </ul>
  </p>
  </body>
</html>
```

For all HTTP Promises the `then` method is set. It gets called whenever the promise has successfully fetched data from MySQL or failed doing so. The `then` method updates the scope property `greetings`. Angular notices the change and updates the view.

For brevity and to reduce the complexity of the examples only few AngularJS features are used. For example, one could use a service for better code reuse.

4.5 The JSON document (DOC) endpoint: /doc/

Arbitrary JSON documents can be stored and retrieved using the DOC endpoint. There is no limit to the nesting level of the documents. All documents have a unique ID and a revision number. Documents can be read, inserted, replaced as a whole and deleted. Search is based on key-document semantics.

HTTP methods are mapped to corresponding data manipulation operations.

- Depending on the URL, a HTTP `PUT` either creates a SQL table to hold documents or inserts a document to a table.
- A `GET` request is used to fetch a document.
- Documents are removed using `DELETE` HTTP requests. A DELETE request can also be used to drop a table and with it all documents it contains.

By default dcuments are stored in BLOB columns of underlying relational tables. The choice of the SQL column type determines the maximum size of the JSON document, other column types can be configured. Appropriate tables can be created and removed through the HTTP DOC endpoint. Other data defintion operations, including modifying the tables that hold documents, are not possible.

JSON is stored "as-is". Whitespace and formatting may not be preserved but JSON datatypes are. Because no mapping between relational tables, columns and SQL data types happens no mismatch between JSON data types and SQL data types can occur.

The only supported character set is UTF-8.

4.5.1 API overview

The DOC endpoint API lets users work with JSON documents. HTTP(S) GET, PUT and DELETE methods can be used to work with documents. All messages exchanged between the client and the server use the JSON format. Like all other endpoints, the DOC endpoint supports the UTF-8 charset only.

Requests for the DOC endpoint must begin with the URL pattern: `protocol://host:port/doc/database/`. The endpoints URL prefix (default: `/doc/`) is configured with the system variable `myhttp_document_url_prefix`. Including a default database in the URL is mandatory but it is valid to use an empty string: `protocol://host:port/doc//`. In case of an empty string the HTTP Plugin defaults to `myhttp_default_db`. Valid URLs are then followed by command specific elements.

For example, the URL pattern to create a new table `blog_posts` for storing documents is: `protocol://host:port/doc/database/table`.

```
shell> curl -X PUT --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/blog_posts'

{
  "info": "Table created"
}
```

The PUT method is used for creating tables, inserting documents and updating documents. Every document has a unique identifier and a revision counter. Clients must provide the document identifier but not the revision number when inserting documents into tables. The HTTP Plugin is using a `VARCHAR(36)` to store the identifier. This allows you to use identifiers with self-explaining values such as `doc_api_overview` in the example below:

```
shell> curl -d '{"when": "2014-09-09 15:22", "title": "API overview", "content": "..."}'
  -X PUT --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/blog_posts/doc_api_overview'

{
  "info": "Document added"
}
```

The revision counter is automatically added to documents upon their creation for optimistic locking. When documents are retrieved using a GET request, they contain two special members added by the HTTP plugin. The `_id` member contains the identifier value and `_rev` is the revision counter.

```
shell> curl --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/blog_posts/doc_api_overview'


{
  "_id": "doc_api_overview",
  "_rev": 1,
  "when": "2014-09-09 15:22",
  "title": "API overview",
  "content": "..."
}
```

The revision counter is a tribute to the stateless nature of the HTTP protocol. If two clients fetch the same document, one updates it and then saves it back to MySQL, the revision counter is incremented. Should the second client also update the document and attempt to write it back to the database, then the database can compare the revision counter the second client provides with the revision stored in the database and detect a possible conflict. Please note, this approach requires a cooperative client which does not modify the revision number.

```
shell>  curl  -d '{"_id":"doc_api_overview","_rev":1,"when":
  "2014-09-09 16:35", "title": "API overview",
  "content": "Optimistic locking" }'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/blog_posts/doc_api_overview'

{
  "info": "Document updated"
}

shell> curl  -d '{"_id":"doc_api_overview","_rev":1,"when":
  "2014-09-09 16:35", "title": "API overview",
  "content": "Optimistic locking" }'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/blog_posts/doc_api_overview'
```

```
{
  "errno": 2000,
  "error": "Update failed. Your revision does not match the current revision"
}
```

Use a HTTP DELETE request to remove a single document or all documents of a table. If the URL contains a document identifier, then the named document will be removed.

```
shell> curl -v
  -X DELETE --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/blog_posts/doc_api_overview'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> DELETE /doc/myhttp/blog_posts/doc_api_overview HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 28
< Pragma: no-cache
< Content-Type: application/json
<
{
  "info": "Document removed"
}
```

The DOC endpoint internally maps client requests to SQL statements internally, just like all other endoints of the HTTP Plugin. SQL sets the limits of the endpoints functionality. Unlike the other endpoints, the DOC endpoint aims to hide SQL details from the user. For example, it does not return meta data information about SQL execution results, such as `affected_rows` but a human readable message like `Document removed` instead.

4.5.2 HTTP methods, headers and status codes

The following HTTP headers are set by the DOC endpoint. The headers do not differ from those set by the SQL endpoint.

Header	Description
Server	Always given. Can be considered as an API version. For example: `MyHTTP 1.0.0-alpha`
Cache-control	Always given. Always `must-revalidate`
Pragma	Always given. Always `no-cache`
Content-Length	Always given.
Content-Type	Set to `application/json` for `200 OK`, `400 Bad Request`, `401 Unauthorized`. Other replies may or may not contain it.
Connection	Always given. Always `Keep-Alive`.

The DOC endpoint supports GET, PUT and DELETE requests. Other requests will be rejected with code `405 Method Not Allowed`. The reply to such a rejected request contains no content. No error message is given.

All resources are protected with HTTP basic authentication. Should HTTP basic authentication fail or not be used with the request at all, the server replies with code `401 Unauthorized` and the JSON error message `{"errno":1045, "sqlstate":"28000","error":"401 Unauthorized"}`.

The HTTP plugin replies with code `400 Bad Request` and no content to any GET, PUT or DELETE request that has a malformed URL. Valid request URLs depend on the request method. For example, a GET request URL is malformed if it does not follow the URL pattern: `protocol://host:port/doc/database/table/` or `protocol://host:port/doc/database/table/document_id` (assuming `myhttp_document_url_prefix=/doc/`).

General errors may also return code `400 Bad Request` but are accomplished with a JSON error document that gives more details about the nature of the error.

4.5.2.1 PUT - create tables, insert documents, update documents

The HTTP PUT method can be used to create tables that store documents, to insert documents in such tables and to replace documents. The URL pattern to create a new table is: `protocol:/host:port/doc/database/table`. To create a table, issue a PUT request that follows the URL pattern but do not send any data with the PUT request to the server. The server will reply with `201 Created` and a message confirming that the table has been created or use code `400 Bad Request` followed by an error message to indicate a failure.

```
shell> curl -X PUT --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/json_types'

{
  "info": "Table created"
}

shell> curl -X PUT -v
  --user basic_auth_user:basic_auth_passwd
    --url 'http://127.0.0.1:8080/doc/myhttp/json_types'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> PUT /doc/myhttp/json_types HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 400 Bad Request
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 47
< Pragma: no-cache
< Content-Type: application/json
<
{
  "errno": 2000,
  "error": "Table already exists."
}
```

Creating and droping a table are the only two data definition operations that can be carried out through the DOC endpoint. A table created by the DOC endpoint has three columns. The primary key is compound of the columns `_id` and `_rev`. The column `_id` is a `VARCHAR(36)` and `_rev` is an unsigned integer. The HTTP plugin implementation assumes the use of these SQL column types. Do not change them.

```
CREATE TABLE `json_types` (
  `_id` VARCHAR(36) NOT NULL,
  `_rev` BIGINT(20) UNSIGNED NOT NULL DEFAULT '0',
  `_extra` BLOB NOT NULL,
  PRIMARY KEY (`_id`,`_rev`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

The DOC endpoint stores JSON documents in the `_extra` column. Any text column can be used, the default is `BLOB`. The maximum length of the `_extra` column determines the size of the JSON documents you can store. A BLOB column of an InnoDB table can hold upto 16MB of data. To prevent users from inserting large documents or to allow storing larger documents, change the column type.

A HTTP PUT request to create a table must not send any data. If any data is send from the client to the HTTP plugin, the plugin assumes that a new document shall be stored or an existing one is to be replaced. The URL pattern to insert or replace documents is: `protocol://server:port/doc/database/table/id`. Failing to include a document identifier value, results in an error. Like for any other error, the return code is set to `400 Bad Request`.

```
shell> curl
  -d '{"json_null": null, "json_string": "string",
      "json_number": 123.456, "json_bool": true}'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/json_types'

{
  "errno": 2000,
  "error": "The request URL must include a document id"
}
```

The data send together with a PUT request in order to insert or replace a JSON document to the database must be a valid JSON object. Arbitrary text, any invalid JSON or any JSON that is not an object will be rejected.

```
shell> curl
  -d 'No JSON, no fun'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/json_types/1'

{
  "errno": 2000,
  "error": "Invalid JSON"
}

shell> curl
  -d '[1,2 ]'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/json_types/1'
```

```
{
  "errno": 2000,
  "error": "Must be a JSON object"
}
```

Only a PUT request that follows the URL pattern for inserting or replacing objects and provides a valid JSON object with the PUT request, results in a new JSON document being added to the database. Furthermore, the JSON object to be inserted must not contain a top level member of the name `_rev`. The presence of a top level `_rev` member indicates that you want to update an already existing document.

Below is an example of a successful document insertion. The request follows the URL pattern and the PUT request includes a valid JSON object . The server replies with code `200 OK` and a message confirming the document has been stored.

```
shell> curl
  -d '{"json_null": null, "json_string": "string",
      "json_number": 123.456, "json_bool": true}'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/json_types/1'

{
  "info": "Document added."
}
```

A document to be inserted should not contain a top level member of the name `_id`. The document identifier value of a new document is set through the URL only. A top level member of the name `_id` will be removed from the JSON object you try to insert. If `_id` is the only member of the JSON object you attempt to insert, then the plugin replies with an error message stating that the object is empty, which is because the `_id` member was implicitly removed.

```
shell> curl
  -d '{"_id": "1"}'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/json_types/2'

{
  "errno": 2000,
  "error": "Empty JSON document"
}
```

To replace and hereby update a document issue a PUT request with the updated document including the special document member `_rev`, which holds the revision of the document. The revision is used for optimistic locking to detect conflicting updates from different clients. When fetching a document from the database, the database ensures the document identifier and revision are included in the document as the `_id` respectively `_rev` object members.

```
shell> curl --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/json_types/1'

{
  "_id": "1",
  "_rev": 1,
  "json_bool": true,
  "json_null": null,
```

```
  "json_string": "string",
  "json_number": 123.456
}
```

Clients should never change the special `_rev` member of a document. The database will increment the revision number when needed, for example, in case of a successful update.

```
shell> curl -d '{"_id":"1","_rev":1,"msg": "first client" }'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/json_types/1'

{
  "info": "Document updated."
}

shell> curl --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/json_types/1'

{
  "_id": "1",
  "_rev": 2,
  "msg": "first client"
}
```

Assume two clients fetch a document to change it. When the first client writes it back to database, the database compares the revision number with the highest revision number known for the document. If the clients documents revision number matches, then the client made his changes to the latest version of the document and the update succeeds. In case of an successful update, the server sends `200 OK` together with a message that confirms the change. As part of the update, the database increases the revision number. See above for an example.

Should the second client also try to update the document, the database detects that the client based his updates on a no longer existing revision and rejects the change to avoid overwriting and loosing newer revisions. It is now up to the second client to solve the conflict, for example, by fetching the latest revision and trying the update again after merging the revisions.

```
shell>  curl -d '{"_id":"1","_rev":1,"msg": "second client" }'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/json_types/1'

{
  "errno": 2000,
  "error": "Update failed. Your revision does not match the current revision."
}
```

The below table summarizes the three actions that can be carried out using the HTTP PUT method.

Action	URL	Content
Creates a table `table`.	`.../doc/database/table`	None
Adds a new document.	`.../doc/database/table/id`	• Non empty, valid JSON object. Special member `_id` not allowed and will be removed prior to inserting.

Action	URL	Content
		• JSON object does not have a top level member `_rev`. `_rev` is a reserved special name for the document revision.
Replaces a document if the given revision still exists.	`.../doc/database/table/id`	• Non empty, valid JSON object. Special member `_id` must match document identifier from the URL. • JSON object does have a top level member `_rev` which holds the revision number of the document.

4.5.2.2 GET - fetching documents

The DOC endpoint offers key document search capabilities. A HTTP GET request can be used to fetch a document identified by its identifier value from a table or all documents from a table.

To demonstrate the feature, create a table that holds two documents.

```
shell> curl -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/hello'

{
  "info": "Table created."
}

shell> curl -d '{"msg": "Hello"}'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/hello/first_word'

{
  "info": "Document added."
}

shell> curl -d '{"msg": "world."}'
  -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/hello/second_word'

{
  "info": "Document added."
}
```

A GET request following the URL pattern `protocol://host:port/doc/database/table/id` will make the HTTP plugin search for a document with the identifiers value `id` in the table `table` from the schema `database`. If the document exists, it is returned and code `200 OK` is used for the HTTP reply. The returned document differs from the one inserted into the database. It has two additional top level object properties `_id` and `_rev`. The `_id` member is a string with the document identifier value. The `_rev` member is the revision number of the document. The revision is used for optimistic locking. Details are given with the description of the PUT method.

```
shell> curl -v --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/hello/first_word'

* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> GET /doc/myhttp/hello/first_word HTTP/1.1
```

```
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 45
< Pragma: no-cache
< Content-Type: application/json
<
{
  "_id": "first_word",
  "_rev": 1,
  "msg": "Hello"
}
```

If the requested document does not exist, the HTTP plugin returns `404 Not Found`. No content is sent with the reply.

```
shell> curl -v --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/hello/be_magic'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> GET /doc/myhttp/hello/be_magic HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 0
< Pragma: no-cache
```

A GET request using the URL pattern `protocol://host:port/doc/database/table/` returns all documents from the table in question. If there are any documents in the table, the return code is `200 OK` otherwise it is `404 Not Found`.

It is not possible to sort results. Documents will be returned in arbitrary order. The order may may coincident with the insertion order, as it is the case with the example, but this is a random side effect. Do not rely on any order.

```
shell> curl -v --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/hello/'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> GET /doc/myhttp/hello/ HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
```

```
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 106
< Pragma: no-cache
<
{
  "hello": [
    {
      "_id": "first_word",
      "_rev": 1,
      "msg": "Hello"
    },
    {
      "_id": "second_word",
      "_rev": 1,
      "msg": "world."
    }
  ]
}
```

Note the slash after the table name at the end of the URL pattern. The slash must be included, otherwise the request is considered invalid. An invalid request results in an empty code `400 Bad Request` reply.

```
shell> curl -v --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/hello'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> GET /doc/myhttp/hello HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 400 Bad Request
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 0
< Pragma: no-cache
< Content-Type: application/json
<
```

Please note, the HTTP plugins DOC endpoint does not check the structure of the tables it works with. Trying to fetch documents from a table that does not exist or does not have the required columns to store and read documents returns an error with code `400 Bad Request`:

```
shell> curl --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/simple/'

{
  "errno": 1054,
```

```
  "error": "Unknown column '_id' in 'field list'"
}
```

The DOC endpoint also does not validate the data it reads. Using a standard MySQL client and SQL, it is possible to manipulate the contents of a document table in an arbitrary way. For example, one could use SQL to insert other data but JSON. A read request issued through the DOC endpoint will not validate the contents and forward them to the client. However, no such problems occur when only the DOC endpoint is used to manipulate documents.

The following table summarizes the GET method features:

Action	URL	Notes
Fetch one document.	`.../doc/database/table/id`	Fetches document `id`. Returns `404 Not Found` if the document is not found, otherwise `200 OK`.
Fetch all documents from a table.	`.../doc/database/table/`	The result set format is described below. Note the mandatory slash after the table name in the URL pattern.

4.5.2.3 GET - Utility commands

HTTP GET requests can also be used to execute utility commands. The URL pattern for utility commands is: `protocol://host:port/doc/_command`. The HTTP Plugin preview version features one command:

Action	URL	Notes
Generate unique identifiers	`.../doc/_uuids`	Supports a count parameter. Returns JSON resultset.

The `_uuids` command returns between 1 and 100 unique identifiers. The unique identifiers can be used when inserting new documents into the database. To do so, a client asks for a list of identifiers and then uses them to create the URL for inserting new documents. The HTTP Plugin generates identifiers using the MySQL SQL function `UUID()`. Please, see the function description for details.

```
shell> curl -v --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/_uuids'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> GET /doc/_uuids HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< X-jsonp-escape: false
< Connection: Keep-Alive
< X-jsonp:
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 50
< Pragma: no-cache
< Content-Type: application/json
<
{
  "uuids": [
```

```
    "4de304bb-3d91-11e4-8e1c-000c2940576d"
  ]
}
```

The function returns one unique identifier by default. The URL parameter `count` can be added to request a different number. The valid range for the parameter is from 1 to 100. Values outside of the range will be silently ignored and limited to the minimum respectively maximum allowed value.

```
shell> curl --user basic_auth_user:basic_auth_passwd --url 'http://127.0.0.1:8080/doc/_uuids?count=3'

{
  "uuids": [
    "234abd93-3d92-11e4-8e1c-000c2940576d",
    "234ac427-3d92-11e4-8e1c-000c2940576d",
    "234ac909-3d92-11e4-8e1c-000c2940576d"
  ]
}
```

The reply code for valid `_uuids` command requests is always `200 OK`.

4.5.2.4 DELETE - removing tables and documents

The DOC endpoint removes documents when it receives a valid HTTP DELETE request. Depending on the URL pattern, the request will cause the deletion of one document or drop the table and with it all documents.

The URL pattern for removing a single document identified by `id` stored in the table `table` from the schema `database` is `protocol://host:port/doc/database/table/id`. If the document is found, the HTTP plugin replies with `200 OK` and a message that confirms the removal. If the document is unknown, an empty reply with code `404 Not Found` is send. An invalid URL pattern causes a code `400 Bad Request` reply with no content.

```
shell> curl --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/hello/second_word'

{
  "_id": "second_word",
  "_rev": 1,
  "msg": "world."
}

shell> curl -v -X DELETE
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/hello/second_word'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> DELETE /doc/myhttp/hello/second_word HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
```

```
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 28
< Pragma: no-cache
< Content-Type: application/json
<
{
  "info": "Document removed"
}

shell> curl -v -X DELETE
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/hello/second_word'

* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> DELETE /doc/myhttp/hello/second_word HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 0
< Pragma: no-cache
<
```

A HTTP DELETE request to drop a table must match the URL pattern `protocol://host:post/doc/database/table/`. If the table to be dropped exists and the user has access to it, the DOC endpoint replies to a DELETE request with code `200 OK` and an info message confirming the removal of the table. An attempt to drop an already dropped table results in a code `404 Not Found` reply with an empty body.

```
shell> curl -v -X DELETE
  --user basic_auth_user:basic_auth_passwd
    --url 'http://127.0.0.1:8080/doc/myhttp/hello'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> DELETE /doc/myhttp/hello HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 25
< Pragma: no-cache
< Content-Type: application/json
<
* Connection #0 to host 127.0.0.1 left intact
{
  "info": "Table dropped"
}
```

The DOC endpoint sends and empty reply with code `404 Not Found` if you try to remove a document from an already dropped table. No hint is given that the table does not exist. Judging from the reply only, a client cannot distinguish this case from an attempt to delete a non-existing document from an existing table.

Invalid DELETE requests or requests that cause errors during their cause code `400 Bad Request` answers. Clients should be prepared to handle errors.

```
shell> curl -v -X DELETE --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/not_allowed/db_and_table'

[...]
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'basic_auth_user'
> DELETE /doc/not_allowed/db_and_table HTTP/1.1
> Authorization: Basic YmFzaWNfYXV0aF91c2VyOmJhc2ljX2F1dGhfcGFzc3dk
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:8080
> Accept: */*
>
< HTTP/1.1 400 Bad Request
< Connection: Keep-Alive
< Cache-control: must-revalidate
* Server MyHTTP 1.0.0-alpha is not blacklisted
< Server: MyHTTP 1.0.0-alpha
< Content-Length: 101
< Pragma: no-cache
< Content-Type: application/json
<
* Connection #0 to host 127.0.0.1 left intact
{
  "errno": 1044,
  "error": "Access denied for user 'http_sql_user'@'127.0.0.1' to database 'not_allowed'"
}
```

Action	URL	Notes
Delete a document.	`.../doc/database/table/id`	Deletes the document `id`. Returns `404 Not Found` if the document was not found, otherwise `200 OK`.
Drop a table.	`.../doc/database/table/`	Returns `200 OK` if the table was found and dropped. Returns `404 Not Found` if the table does not exist.

4.5.3 JSON content formats

The DOC endpoint sends a JSON object in reply to all valid requests. The JSON object has one of three formats. There are resultsets, error messages and info messages.

```
REPLY:
 jsonp_function(answer) |
 answer

jsonp_function:
  string

answer:
  resultset |
```

```
  error |
  info |
  uuids
```

All JSON documents returned are simple, lightweight and easy to parse. Their size is comparable to the documents returned by the CRUD endpoint. They are smaller than those returned by the SQL endpoint, for example, because the resultsets do not contain any meta data information related to the underlying SQL storage. Please note that JSON unicode encoding for special characters can significantly grow the storage requirements of your documents. Documents are stored "as-is". They are not serialized into any kind of binary format for storage.

Different requests cause different replies and reply formats. Clients can consider their request and the return code to determine what kind of JSON document to expect in reply, if any. The following table gives and overview.

HTTP return code	Content	Methods	Details
401 Unauthorized	Error document	All	Wrong HTTP basic authentication credentials
400 Bad Request	Error document	All	Check error message.
405 Method Not Allowed	No content	Any but GET, PUT, DELETE	
200 OK	Result document	GET	In reply to read documents
200 OK	UUIDs document	GET	In reply to `_uuids` command
200 OK	Info document	PUT used to insert or replace documents, DELETE	
201 Created	Info document	PUT used for table creation	
404 Not Found	No content	GET, DELETE	

4.5.4 JSON result document

The DOC endpoint resultset contains one or more JSON documents, depending on the request used. If the client request has generated multiple documents, the resultset is a document list. The document list holds all the documents produced. A document list is a JSON object with one member. The members name is the table from which the documents have been fetched. The member value is an array of documents.

```


resultset:
  document_list |
  document

document_list:
{
  "table_name":
    [
      document (, document ...)
    ]
}

document:
{
```

```
  "_id" : document_id,
  "_rev" : revision,
  user_input
}

document_id: string
revision: number

user_input:
  member : value
  (, member : value ...)
```

A document from a resultset is an enriched variant of the document stored in the database. The document contains all data once stored in the database plus two additional top level members. The special member `_id` holds the document identifier value. It is always a string. The special member `_rev` is the revision number of the document.

Resultsets can be expected in reply to a GET request if the return code is `200 OK`

4.5.5 JSON error document

Clients can expect error documents when the return code is `400 Bad Request` or `401 Unauthorized`. Errors may be returned in reply to any valid request method. The error document is a JSON object with two members: `errno`, `error`.

```
error:
{
 "errno": int,
 "error": string
}
```

The error number `errno` and the error description `error` may match MySQL server error codes and messages. All HTTP plugin endpoints use SQL internally to carry out their work. In some cases the DOC endpoint may copy SQL error messages into the error document. In other cases, it sets its own error code and message.

```
shell> curl -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/unkown_db/unknown_table'

{
  "errno": 1044,
  "error": "Access denied for user 'http_sql_user'@'127.0.0.1' to database 'unkown_db'"
}
```

4.5.6 JSON info document

The info document contains a short message which confirms the execution of the a request. It is sent in reply to PUT and DELETE requests if the return code is `200 OK` or `201 Created`.

```
info:
{
 "info": string,
}
```

Below is an example of the message returned in reply to a successful table creation. The message itself adds no information to the reply because the client can determine the outcome of the request from the

return code `201 Created`. The info document is an alternative and convenient way to inform the client of a request outcome without having to check the result code which may not be always possible.

```
shell> curl -X PUT
  --user basic_auth_user:basic_auth_passwd
  --url 'http://127.0.0.1:8080/doc/myhttp/new_table'

{
  "info": "Table created"
}
```

4.5.7 JSON UUIDs document

The special command `_uuids` returns a UUID document with one or more unique identifiers. The returned JSON object has one property called `uuids`. The object member holds a list of strings, which are the unique identifiers.

```
uuids:
{
  "uuids":
    [
      string (, string ...)
    ]
}
```