# mysql DT

2.18.1 • Public • Published 2 years ago

Readme

Explore BETA

4 Dependencies

6,553 Dependents

65 Versions

# mysql

npm v2.18.1 downloads 3.61M/month node >= 0.6 travis passing windows success coverage 98%

## Table of Contents

# Install

This is a **Node.js** module available through the **npm registry**.

Before installing, **download and install Node.js**. Node.js 0.6 or higher is required.

Installation is done using the **`npm install` command**:

```
$ npm install mysql
```

For information about the previous 0.9.x releases, visit the **v0.9 branch**.

Sometimes I may also ask you to install the latest version from Github to check if a bugfix is working. In this case, please do:

```
$ npm install mysqljs/mysql
```

# Introduction

This is a node.js driver for mysql. It is written in JavaScript, does not require compiling, and is 100% MIT licensed.

Here is an example on how to use it:

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'me',
  password : 'secret',
  database : 'my_db'
});

connection.connect();

connection.query('SELECT 1 + 1 AS solution', function (error, results,
  if (error) throw error;
  console.log('The solution is: ', results[0].solution);
});

connection.end();
```

From this example, you can learn the following:

- Every method you invoke on a connection is queued and executed in sequence.
- Closing the connection is done using `end()` which makes sure all remaining queries are executed before sending a quit packet to the mysql server.

# Contributors

Thanks goes to the people who have contributed code to this module, see the **GitHub Contributors page**.

Additionally I'd like to thank the following people:

- Andrey Hristov (Oracle) - for helping me with protocol questions.
- Ulf Wendel (Oracle) - for helping me with protocol questions.

## Sponsors

The following companies have supported this project financially, allowing me to spend more time on it (ordered by time of contribution):

- Transloadit (my startup, we do file uploading & video encoding as a service, check it out)
- Joyent
- pinkbike.com
- Holiday Extras (they are hiring)
- Newscope (they are hiring)

## Community

If you'd like to discuss this module, or ask questions about it, please use one of the following:

- **Mailing list**: https://groups.google.com/forum/#!forum/node-mysql
- **IRC Channel**: #node.js (on freenode.net, I pay attention to any message including the term `mysql`)

## Establishing connections

The recommended way to establish a connection is this:

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
  host     : 'example.org',
  user     : 'bob',
```

```
  password : 'secret'
});

connection.connect(function(err) {
  if (err) {
    console.error('error connecting: ' + err.stack);
    return;
  }

  console.log('connected as id ' + connection.threadId);
});
```

However, a connection can also be implicitly established by invoking a query:

```
var mysql      = require('mysql');
var connection = mysql.createConnection(...);

connection.query('SELECT 1', function (error, results, fields) {
  if (error) throw error;
  // connected!
});
```

Depending on how you like to handle your errors, either method may be appropriate. Any type of connection error (handshake or network) is considered a fatal error, see the **Error Handling** section for more information.

## Connection options

When establishing a connection, you can set the following options:

- `host` : The hostname of the database you are connecting to. (Default: `localhost` )
- `port` : The port number to connect to. (Default: `3306` )
- `localAddress` : The source IP address to use for TCP connection. (Optional)

- `socketPath` : The path to a unix domain socket to connect to. When used `host` and `port` are ignored.
- `user` : The MySQL user to authenticate as.
- `password` : The password of that MySQL user.
- `database` : Name of the database to use for this connection (Optional).
- `charset` : The charset for the connection. This is called "collation" in the SQL-level of MySQL (like `utf8_general_ci` ). If a SQL-level charset is specified (like `utf8mb4` ) then the default collation for that charset is used. (Default: `'UTF8_GENERAL_CI'` )
- `timezone` : The timezone configured on the MySQL server. This is used to type cast server date/time values to JavaScript `Date` object and vice versa. This can be `'local'` , `'Z'` , or an offset in the form `+HH:MM` or `-HH:MM` . (Default: `'local'` )
- `connectTimeout` : The milliseconds before a timeout occurs during the initial connection to the MySQL server. (Default: `10000` )
- `stringifyObjects` : Stringify objects instead of converting to values. See issue #501. (Default: `false` )
- `insecureAuth` : Allow connecting to MySQL instances that ask for the old (insecure) authentication method. (Default: `false` )
- `typeCast` : Determines if column values should be converted to native JavaScript types. (Default: `true` )
- `queryFormat` : A custom query format function. See Custom format.
- `supportBigNumbers` : When dealing with big numbers (BIGINT and DECIMAL columns) in the database, you should enable this option (Default: `false` ).
- `bigNumberStrings` : Enabling both `supportBigNumbers` and `bigNumberStrings` forces big numbers (BIGINT and DECIMAL columns) to be always returned as JavaScript String objects (Default: `false` ). Enabling `supportBigNumbers` but leaving `bigNumberStrings` disabled will return big numbers as String objects only when they cannot be accurately represented with JavaScript Number objects (which happens when they exceed the [-2^53, +2^53] range), otherwise they will be returned as Number objects. This option is ignored if `supportBigNumbers` is disabled.
- `dateStrings` : Force date types (TIMESTAMP, DATETIME, DATE) to be returned as strings rather than inflated into JavaScript Date objects. Can be `true` / `false` or an array of type names to keep as strings. (Default: `false` )
- `debug` : Prints protocol details to stdout. Can be `true` / `false` or an array of packet type names that should be printed. (Default: `false` )

- `trace` : Generates stack traces on `Error` to include call site of library entrance ("long stack traces"). Slight performance penalty for most calls. (Default: `true` )
- `localInfile` : Allow `LOAD DATA INFILE` to use the `LOCAL` modifier. (Default: `true` )
- `multipleStatements` : Allow multiple mysql statements per query. Be careful with this, it could increase the scope of SQL injection attacks. (Default: `false` )
- `flags` : List of connection flags to use other than the default ones. It is also possible to blacklist default ones. For more information, check Connection Flags.
- `ssl` : object with ssl parameters or a string containing name of ssl profile. See SSL options.

In addition to passing these options as an object, you can also use a url string. For example:

```
var connection = mysql.createConnection('mysql://user:pass@host/db?deb
```

Note: The query values are first attempted to be parsed as JSON, and if that fails assumed to be plaintext strings.

### SSL options

The `ssl` option in the connection options takes a string or an object. When given a string, it uses one of the predefined SSL profiles included. The following profiles are included:

- `"Amazon RDS"` : this profile is for connecting to an Amazon RDS server and contains the certificates from https://rds.amazonaws.com/doc/rds-ssl-ca-cert.pem and https://s3.amazonaws.com/rds-downloads/rds-combined-ca-bundle.pem

When connecting to other servers, you will need to provide an object of options, in the same format as **tls.createSecureContext**. Please note the arguments expect a string of the certificate, not a file name to the certificate. Here is a simple example:

```
var connection = mysql.createConnection({
  host : 'localhost',
  ssl  : {
    ca : fs.readFileSync(__dirname + '/mysql-ca.crt')
  }
```

```
});
```

You can also connect to a MySQL server without properly providing the appropriate CA to trust. *You should not do this.*

```
var connection = mysql.createConnection({
  host : 'localhost',
  ssl  : {
    // DO NOT DO THIS
    // set up your ca correctly to trust the connection
    rejectUnauthorized: false
  }
});
```

### Connection flags

If, for any reason, you would like to change the default connection flags, you can use the connection option `flags`. Pass a string with a comma separated list of items to add to the default flags. If you don't want a default flag to be used prepend the flag with a minus sign. To add a flag that is not in the default list, just write the flag name, or prefix it with a plus (case insensitive).

```
var connection = mysql.createConnection({
  // disable FOUND_ROWS flag, enable IGNORE_SPACE flag
  flags: '-FOUND_ROWS,IGNORE_SPACE'
});
```

The following flags are available:

- `COMPRESS` - Enable protocol compression. This feature is not currently supported by the Node.js implementation so cannot be turned on. (Default off)
- `CONNECT_WITH_DB` - Ability to specify the database on connection. (Default on)
- `FOUND_ROWS` - Send the found rows instead of the affected rows as `affectedRows`. (Default on)

- `IGNORE_SIGPIPE` - Don't issue SIGPIPE if network failures. This flag has no effect on this Node.js implementation. (Default on)
- `IGNORE_SPACE` - Let the parser ignore spaces before the `(` in queries. (Default on)
- `INTERACTIVE` - Indicates to the MySQL server this is an "interactive" client. This will use the interactive timeouts on the MySQL server and report as interactive in the process list. (Default off)
- `LOCAL_FILES` - Can use `LOAD DATA LOCAL`. This flag is controlled by the connection option `localInfile`. (Default on)
- `LONG_FLAG` - Longer flags in Protocol::ColumnDefinition320. (Default on)
- `LONG_PASSWORD` - Use the improved version of Old Password Authentication. (Default on)
- `MULTI_RESULTS` - Can handle multiple resultsets for queries. (Default on)
- `MULTI_STATEMENTS` - The client may send multiple statement per query or statement prepare (separated by `;`). This flag is controlled by the connection option `multipleStatements`. (Default off)
- `NO_SCHEMA`
- `ODBC` Special handling of ODBC behaviour. This flag has no effect on this Node.js implementation. (Default on)
- `PLUGIN_AUTH` - Uses the plugin authentication mechanism when connecting to the MySQL server. This feature is not currently supported by the Node.js implementation so cannot be turned on. (Default off)
- `PROTOCOL_41` - Uses the 4.1 protocol. (Default on)
- `PS_MULTI_RESULTS` - Can handle multiple resultsets for execute. (Default on)
- `REMEMBER_OPTIONS` - This is specific to the C client, and has no effect on this Node.js implementation. (Default off)
- `RESERVED` - Old flag for the 4.1 protocol. (Default on)
- `SECURE_CONNECTION` - Support native 4.1 authentication. (Default on)
- `SSL` - Use SSL after handshake to encrypt data in transport. This feature is controlled though the `ssl` connection option, so the flag has no effect. (Default off)
- `SSL_VERIFY_SERVER_CERT` - Verify the server certificate during SSL set up. This feature is controlled though the `ssl.rejectUnauthorized` connection option, so the flag has no effect. (Default off)
- `TRANSACTIONS` - Asks for the transaction status flags. (Default on)

# Terminating connections

There are two ways to end a connection. Terminating a connection gracefully is done by calling the `end()` method:

```
connection.end(function(err) {
  // The connection is terminated now
});
```

This will make sure all previously enqueued queries are still before sending a `COM_QUIT` packet to the MySQL server. If a fatal error occurs before the `COM_QUIT` packet can be sent, an `err` argument will be provided to the callback, but the connection will be terminated regardless of that.

An alternative way to end the connection is to call the `destroy()` method. This will cause an immediate termination of the underlying socket. Additionally `destroy()` guarantees that no more events or callbacks will be triggered for the connection.

```
connection.destroy();
```

Unlike `end()` the `destroy()` method does not take a callback argument.

## Pooling connections

Rather than creating and managing connections one-by-one, this module also provides built-in connection pooling using `mysql.createPool(config)`. **Read more about connection pooling**.

Create a pool and use it directly:

```
var mysql = require('mysql');
var pool  = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bob',
  password        : 'secret',
```

```
  database        : 'my_db'
});

pool.query('SELECT 1 + 1 AS solution', function (error, results, fielc
  if (error) throw error;
  console.log('The solution is: ', results[0].solution);
});
```

This is a shortcut for the `pool.getConnection() -> connection.query() -> connection.release()` code flow. Using `pool.getConnection()` is useful to share connection state for subsequent queries. This is because two calls to `pool.query()` may use two different connections and run in parallel. This is the basic structure:

```
var mysql = require('mysql');
var pool  = mysql.createPool(...);

pool.getConnection(function(err, connection) {
  if (err) throw err; // not connected!

  // Use the connection
  connection.query('SELECT something FROM sometable', function (error,
    // When done with the connection, release it.
    connection.release();

    // Handle error after the release.
    if (error) throw error;

    // Don't use the connection here, it has been returned to the pool
  });
});
```

If you would like to close the connection and remove it from the pool, use `connection.destroy()` instead. The pool will create a new connection the next time one is needed.

Connections are lazily created by the pool. If you configure the pool to allow up to 100 connections, but only ever use 5 simultaneously, only 5 connections will be made. Connections are also cycled round-robin style, with connections being taken from the top of the pool and returning to the bottom.

When a previous connection is retrieved from the pool, a ping packet is sent to the server to check if the connection is still good.

# Pool options

Pools accept all the same **options as a connection**. When creating a new connection, the options are simply passed to the connection constructor. In addition to those options pools accept a few extras:

- `acquireTimeout` : The milliseconds before a timeout occurs during the connection acquisition. This is slightly different from `connectTimeout` , because acquiring a pool connection does not always involve making a connection. If a connection request is queued, the time the request spends in the queue does not count towards this timeout. (Default: `10000` )
- `waitForConnections` : Determines the pool's action when no connections are available and the limit has been reached. If `true` , the pool will queue the connection request and call it when one becomes available. If `false` , the pool will immediately call back with an error. (Default: `true` )
- `connectionLimit` : The maximum number of connections to create at once. (Default: `10` )
- `queueLimit` : The maximum number of connection requests the pool will queue before returning an error from `getConnection` . If set to `0` , there is no limit to the number of queued connection requests. (Default: `0` )

# Pool events

### acquire

The pool will emit an `acquire` event when a connection is acquired from the pool. This is called after all acquiring activity has been performed on the connection, right before the connection is handed to the callback of the acquiring code.

```
pool.on('acquire', function (connection) {
  console.log('Connection %d acquired', connection.threadId);
});
```

### connection

The pool will emit a `connection` event when a new connection is made within the pool. If you need to set session variables on the connection before it gets used, you can listen to the `connection` event.

```
pool.on('connection', function (connection) {
  connection.query('SET SESSION auto_increment_increment=1')
});
```

### enqueue

The pool will emit an `enqueue` event when a callback has been queued to wait for an available connection.

```
pool.on('enqueue', function () {
  console.log('Waiting for available connection slot');
});
```

### release

The pool will emit a `release` event when a connection is released back to the pool. This is called after all release activity has been performed on the connection, so the connection will be listed as free at the time of the event.

```
pool.on('release', function (connection) {
  console.log('Connection %d released', connection.threadId);
});
```

## Closing all the connections in a pool

When you are done using the pool, you have to end all the connections or the Node.js event loop will stay active until the connections are closed by the MySQL server. This is typically done if the pool is used in a script or when trying to gracefully shutdown a server. To end all the connections in the pool, use the `end` method on the pool:

```
pool.end(function (err) {
  // all connections in the pool have ended
});
```

The `end` method takes an *optional* callback that you can use to know when all the connections are ended.

**Once `pool.end` is called, `pool.getConnection` and other operations can no longer be performed.** Wait until all connections in the pool are released before calling `pool.end`. If you use the shortcut method `pool.query`, in place of `pool.getConnection` → `connection.query` → `connection.release`, wait until it completes.

`pool.end` calls `connection.end` on every active connection in the pool. This queues a `QUIT` packet on the connection and sets a flag to prevent `pool.getConnection` from creating new connections. All commands / queries already in progress will complete, but new commands won't execute.

## PoolCluster

PoolCluster provides multiple hosts connection. (group & retry & selector)

```
// create
var poolCluster = mysql.createPoolCluster();

// add configurations (the config is a pool config object)
poolCluster.add(config); // add configuration with automatic name
```

```
poolCluster.add('MASTER', masterConfig); // add a named configuration
poolCluster.add('SLAVE1', slave1Config);
poolCluster.add('SLAVE2', slave2Config);

// remove configurations
poolCluster.remove('SLAVE2'); // By nodeId
poolCluster.remove('SLAVE*'); // By target group : SLAVE1-2

// Target Group : ALL(anonymous, MASTER, SLAVE1-2), Selector : round-r
poolCluster.getConnection(function (err, connection) {});

// Target Group : MASTER, Selector : round-robin
poolCluster.getConnection('MASTER', function (err, connection) {});

// Target Group : SLAVE1-2, Selector : order
// If can't connect to SLAVE1, return SLAVE2. (remove SLAVE1 in the cl
poolCluster.on('remove', function (nodeId) {
  console.log('REMOVED NODE : ' + nodeId); // nodeId = SLAVE1
});

// A pattern can be passed with *  as wildcard
poolCluster.getConnection('SLAVE*', 'ORDER', function (err, connectior

// The pattern can also be a regular expression
poolCluster.getConnection(/^SLAVE[12]$/, function (err, connection) {

// of namespace : of(pattern, selector)
poolCluster.of('*').getConnection(function (err, connection) {});

var pool = poolCluster.of('SLAVE*', 'RANDOM');
pool.getConnection(function (err, connection) {});
pool.getConnection(function (err, connection) {});
pool.query(function (error, results, fields) {});
```

```
// close all connections
poolCluster.end(function (err) {
  // all connections in the pool cluster have ended
});
```

### PoolCluster options

- `canRetry` : If `true` , `PoolCluster` will attempt to reconnect when connection fails. (Default: `true` )
- `removeNodeErrorCount` : If connection fails, node's `errorCount` increases. When `errorCount` is greater than `removeNodeErrorCount` , remove a node in the `PoolCluster` . (Default: `5` )
- `restoreNodeTimeout` : If connection fails, specifies the number of milliseconds before another connection attempt will be made. If set to `0` , then node will be removed instead and never re-used. (Default: `0` )
- `defaultSelector` : The default selector. (Default: `RR` )
  - RR : Select one alternately. (Round-Robin)
  - RANDOM : Select the node by random function.
  - ORDER : Select the first node available unconditionally.

```
var clusterConfig = {
  removeNodeErrorCount: 1, // Remove the node immediately when connect
  defaultSelector: 'ORDER'
};

var poolCluster = mysql.createPoolCluster(clusterConfig);
```

## Switching users and altering connection state

MySQL offers a changeUser command that allows you to alter the current user and other aspects of the connection without shutting down the underlying socket:

```
connection.changeUser({user : 'john'}, function(err) {
```

```
  if (err) throw err;
});
```

The available options for this feature are:

- `user` : The name of the new user (defaults to the previous one).
- `password` : The password of the new user (defaults to the previous one).
- `charset` : The new charset (defaults to the previous one).
- `database` : The new database (defaults to the previous one).

A sometimes useful side effect of this functionality is that this function also resets any connection state (variables, transactions, etc.).

Errors encountered during this operation are treated as fatal connection errors by this module.

## Server disconnects

You may lose the connection to a MySQL server due to network problems, the server timing you out, the server being restarted, or crashing. All of these events are considered fatal errors, and will have the `err.code = 'PROTOCOL_CONNECTION_LOST'` . See the **Error Handling** section for more information.

Re-connecting a connection is done by establishing a new connection. Once terminated, an existing connection object cannot be re-connected by design.

With Pool, disconnected connections will be removed from the pool freeing up space for a new connection to be created on the next getConnection call.

With PoolCluster, disconnected connections will count as errors against the related node, incrementing the error code for that node. Once there are more than `removeNodeErrorCount` errors on a given node, it is removed from the cluster. When this occurs, the PoolCluster may emit a `POOL_NONEONLINE` error if there are no longer any matching nodes for the pattern. The `restoreNodeTimeout` config can be set to restore offline nodes after a given timeout.

# Performing queries

The most basic way to perform a query is to call the `.query()` method on an object (like a `Connection`, `Pool`, or `PoolNamespace` instance).

The simplest form of `.query()` is `.query(sqlString, callback)`, where a SQL string is the first argument and the second is a callback:

```
connection.query('SELECT * FROM `books` WHERE `author` = "David"', fur
  // error will be an Error if one occurred during the query
  // results will contain the results of the query
  // fields will contain information about the returned results fields
});
```

The second form `.query(sqlString, values, callback)` comes when using placeholder values (see **escaping query values**):

```
connection.query('SELECT * FROM `books` WHERE `author` = ?', ['David']
  // error will be an Error if one occurred during the query
  // results will contain the results of the query
  // fields will contain information about the returned results fields
});
```

The third form `.query(options, callback)` comes when using various advanced options on the query, like **escaping query values**, **joins with overlapping column names**, **timeouts**, and **type casting**.

```
connection.query({
  sql: 'SELECT * FROM `books` WHERE `author` = ?',
  timeout: 40000, // 40s
  values: ['David']
}, function (error, results, fields) {
```

```
  // error will be an Error if one occurred during the query
  // results will contain the results of the query
  // fields will contain information about the returned results fields
});
```

Note that a combination of the second and third forms can be used where the placeholder values are passed as an argument and not in the options object. The `values` argument will override the `values` in the option object.

```
connection.query({
    sql: 'SELECT * FROM `books` WHERE `author` = ?',
    timeout: 40000, // 40s
  },
  ['David'],
  function (error, results, fields) {
    // error will be an Error if one occurred during the query
    // results will contain the results of the query
    // fields will contain information about the returned results fiel
  }
);
```

If the query only has a single replacement character ( `?` ), and the value is not `null` , `undefined` , or an array, it can be passed directly as the second argument to `.query` :

```
connection.query(
  'SELECT * FROM `books` WHERE `author` = ?',
  'David',
  function (error, results, fields) {
    // error will be an Error if one occurred during the query
    // results will contain the results of the query
    // fields will contain information about the returned results fiel
  }
```

```
);
```

## Escaping query values

**Caution** These methods of escaping values only works when the NO_BACKSLASH_ESCAPES SQL mode is disabled (which is the default state for MySQL servers).

In order to avoid SQL Injection attacks, you should always escape any user provided data before using it inside a SQL query. You can do so using the `mysql.escape()`, `connection.escape()` or `pool.escape()` methods:

```
var userId = 'some user provided value';
var sql    = 'SELECT * FROM users WHERE id = ' + connection.escape(use
connection.query(sql, function (error, results, fields) {
  if (error) throw error;
  // ...
});
```

Alternatively, you can use `?` characters as placeholders for values you would like to have escaped like this:

```
connection.query('SELECT * FROM users WHERE id = ?', [userId], functic
  if (error) throw error;
  // ...
});
```

Multiple placeholders are mapped to values in the same order as passed. For example, in the following query `foo` equals `a`, `bar` equals `b`, `baz` equals `c`, and `id` will be `userId`:

```
connection.query('UPDATE users SET foo = ?, bar = ?, baz = ? WHERE id
  if (error) throw error;
```

```
  // ...
});
```

This looks similar to prepared statements in MySQL, however it really just uses the same `connection.escape()` method internally.

**Caution** This also differs from prepared statements in that all `?` are replaced, even those contained in comments and strings.

Different value types are escaped differently, here is how:

- Numbers are left untouched
- Booleans are converted to `true` / `false`
- Date objects are converted to `'YYYY-mm-dd HH:ii:ss'` strings
- Buffers are converted to hex strings, e.g. `X'0fa5'`
- Strings are safely escaped
- Arrays are turned into list, e.g. `['a', 'b']` turns into `'a', 'b'`
- Nested arrays are turned into grouped lists (for bulk inserts), e.g. `[['a', 'b'], ['c', 'd']]` turns into `('a', 'b'), ('c', 'd')`
- Objects that have a `toSqlString` method will have `.toSqlString()` called and the returned value is used as the raw SQL.
- Objects are turned into `key = 'val'` pairs for each enumerable property on the object. If the property's value is a function, it is skipped; if the property's value is an object, toString() is called on it and the returned value is used.
- `undefined` / `null` are converted to `NULL`
- `NaN` / `Infinity` are left as-is. MySQL does not support these, and trying to insert them as values will trigger MySQL errors until they implement support.

This escaping allows you to do neat things like this:

```
var post  = {id: 1, title: 'Hello MySQL'};
var query = connection.query('INSERT INTO posts SET ?', post, functior
  if (error) throw error;
  // Neat!
});
```

```
console.log(query.sql); // INSERT INTO posts SET `id` = 1, `title` = '
```

And the `toSqlString` method allows you to form complex queries with functions:

```
var CURRENT_TIMESTAMP = { toSqlString: function() { return 'CURRENT_TI
var sql = mysql.format('UPDATE posts SET modified = ? WHERE id = ?', [
console.log(sql); // UPDATE posts SET modified = CURRENT_TIMESTAMP() W
```

To generate objects with a `toSqlString` method, the `mysql.raw()` method can be used. This creates an object that will be left un-touched when using in a `?` placeholder, useful for using functions as dynamic values:

**Caution** The string provided to `mysql.raw()` will skip all escaping functions when used, so be careful when passing in unvalidated input.

```
var CURRENT_TIMESTAMP = mysql.raw('CURRENT_TIMESTAMP()');
var sql = mysql.format('UPDATE posts SET modified = ? WHERE id = ?', [
console.log(sql); // UPDATE posts SET modified = CURRENT_TIMESTAMP() W
```

If you feel the need to escape queries by yourself, you can also use the escaping function directly:

```
var query = "SELECT * FROM posts WHERE title=" + mysql.escape("Hello M

console.log(query); // SELECT * FROM posts WHERE title='Hello MySQL'
```

# Escaping query identifiers

If you can't trust an SQL identifier (database / table / column name) because it is provided by a user, you should escape it with `mysql.escapeId(identifier)`,

`connection.escapeId(identifier)` or `pool.escapeId(identifier)` like this:

```
var sorter = 'date';
var sql    = 'SELECT * FROM posts ORDER BY ' + connection.escapeId(sor
connection.query(sql, function (error, results, fields) {
  if (error) throw error;
  // ...
});
```

It also supports adding qualified identifiers. It will escape both parts.

```
var sorter = 'date';
var sql    = 'SELECT * FROM posts ORDER BY ' + connection.escapeId('po
// -> SELECT * FROM posts ORDER BY `posts`.`date`
```

If you do not want to treat `.` as qualified identifiers, you can set the second argument to `true` in order to keep the string as a literal identifier:

```
var sorter = 'date.2';
var sql    = 'SELECT * FROM posts ORDER BY ' + connection.escapeId(sor
// -> SELECT * FROM posts ORDER BY `date.2`
```

Alternatively, you can use `??` characters as placeholders for identifiers you would like to have escaped like this:

```
var userId = 1;
var columns = ['username', 'email'];
var query = connection.query('SELECT ?? FROM ?? WHERE id = ?', [columr
  if (error) throw error;
  // ...
});
```

```
console.log(query.sql); // SELECT `username`, `email` FROM `users` WHE
```

**Please note that this last character sequence is experimental and syntax might change**

When you pass an Object to `.escape()` or `.query()`, `.escapeId()` is used to avoid SQL injection in object keys.

### Preparing Queries

You can use mysql.format to prepare a query with multiple insertion points, utilizing the proper escaping for ids and values. A simple example of this follows:

```
var sql = "SELECT * FROM ?? WHERE ?? = ?";
var inserts = ['users', 'id', userId];
sql = mysql.format(sql, inserts);
```

Following this you then have a valid, escaped query that you can then send to the database safely. This is useful if you are looking to prepare the query before actually sending it to the database. As mysql.format is exposed from SqlString.format you also have the option (but are not required) to pass in stringifyObject and timezone, allowing you provide a custom means of turning objects into strings, as well as a location-specific/timezone-aware Date.

### Custom format

If you prefer to have another type of query escape format, there's a connection configuration option you can use to define a custom format function. You can access the connection object if you want to use the built-in `.escape()` or any other connection function.

Here's an example of how to implement another format:

```
connection.config.queryFormat = function (query, values) {
  if (!values) return query;
  return query.replace(/\:(\w+)/g, function (txt, key) {
    if (values.hasOwnProperty(key)) {
      return this.escape(values[key]);
```

```
    }
    return txt;
  }.bind(this));
};

connection.query("UPDATE posts SET title = :title", { title: "Hello My
```

## Getting the id of an inserted row

If you are inserting a row into a table with an auto increment primary key, you can retrieve the insert id like this:

```
connection.query('INSERT INTO posts SET ?', {title: 'test'}, function
  if (error) throw error;
  console.log(results.insertId);
});
```

When dealing with big numbers (above JavaScript Number precision limit), you should consider enabling `supportBigNumbers` option to be able to read the insert id as a string, otherwise it will throw an error.

This option is also required when fetching big numbers from the database, otherwise you will get values rounded to hundreds or thousands due to the precision limit.

## Getting the number of affected rows

You can get the number of affected rows from an insert, update or delete statement.

```
connection.query('DELETE FROM posts WHERE title = "wrong"', function (
  if (error) throw error;
  console.log('deleted ' + results.affectedRows + ' rows');
})
```

## Getting the number of changed rows

You can get the number of changed rows from an update statement.

"changedRows" differs from "affectedRows" in that it does not count updated rows whose values were not changed.

```
connection.query('UPDATE posts SET ...', function (error, results, fie
  if (error) throw error;
  console.log('changed ' + results.changedRows + ' rows');
})
```

## Getting the connection ID

You can get the MySQL connection ID ("thread ID") of a given connection using the `threadId` property.

```
connection.connect(function(err) {
  if (err) throw err;
  console.log('connected as id ' + connection.threadId);
});
```

## Executing queries in parallel

The MySQL protocol is sequential, this means that you need multiple connections to execute queries in parallel. You can use a Pool to manage connections, one simple approach is to create one connection per incoming http request.

## Streaming query rows

Sometimes you may want to select large quantities of rows and process each of them as they are received. This can be done like this:

```
var query = connection.query('SELECT * FROM posts');
query
  .on('error', function(err) {
    // Handle error, an 'end' event will be emitted after this as well
  })
  .on('fields', function(fields) {
    // the field packets for the rows to follow
  })
  .on('result', function(row) {
    // Pausing the connnection is useful if your processing involves I
    connection.pause();

    processRow(row, function() {
      connection.resume();
    });
  })
  .on('end', function() {
    // all rows have been received
  });
```

Please note a few things about the example above:

- Usually you will want to receive a certain amount of rows before starting to throttle the connection using `pause()`. This number will depend on the amount and size of your rows.
- `pause()` / `resume()` operate on the underlying socket and parser. You are guaranteed that no more `'result'` events will fire after calling `pause()`.
- You MUST NOT provide a callback to the `query()` method when streaming rows.
- The `'result'` event will fire for both rows as well as OK packets confirming the success of a INSERT/UPDATE query.

- It is very important not to leave the result paused too long, or you may encounter `Error: Connection lost: The server closed the connection.` The time limit for this is determined by the net_write_timeout setting on your MySQL server.

Additionally you may be interested to know that it is currently not possible to stream individual row columns, they will always be buffered up entirely. If you have a good use case for streaming large fields to and from MySQL, I'd love to get your thoughts and contributions on this.

### Piping results with Streams

The query object provides a convenience method `.stream([options])` that wraps query events into a **Readable Stream** object. This stream can easily be piped downstream and provides automatic pause/resume, based on downstream congestion and the optional `highWaterMark`. The `objectMode` parameter of the stream is set to `true` and cannot be changed (if you need a byte stream, you will need to use a transform stream, like **objstream** for example).

For example, piping query results into another stream (with a max buffer of 5 objects) is simply:

```
connection.query('SELECT * FROM posts')
  .stream({highWaterMark: 5})
  .pipe(...);
```

## Multiple statement queries

Support for multiple statements is disabled for security reasons (it allows for SQL injection attacks if values are not properly escaped). To use this feature you have to enable it for your connection:

```
var connection = mysql.createConnection({multipleStatements: true});
```

Once enabled, you can execute multiple statement queries like any other query:

```
connection.query('SELECT 1; SELECT 2', function (error, results, fielc
  if (error) throw error;
  // `results` is an array with one element for every statement in the
  console.log(results[0]); // [{1: 1}]
  console.log(results[1]); // [{2: 2}]
});
```

Additionally you can also stream the results of multiple statement queries:

```
var query = connection.query('SELECT 1; SELECT 2');

query
  .on('fields', function(fields, index) {
    // the fields for the result rows that follow
  })
  .on('result', function(row, index) {
    // index refers to the statement this result belongs to (starts at
  });
```

If one of the statements in your query causes an error, the resulting Error object contains a `err.index` property which tells you which statement caused it. MySQL will also stop executing any remaining statements when an error occurs.

Please note that the interface for streaming multiple statement queries is experimental and I am looking forward to feedback on it.

## Stored procedures

You can call stored procedures from your queries as with any other mysql driver. If the stored procedure produces several result sets, they are exposed to you the same way as the results for multiple statement queries.

# Joins with overlapping column names

When executing joins, you are likely to get result sets with overlapping column names.

By default, node-mysql will overwrite colliding column names in the order the columns are received from MySQL, causing some of the received values to be unavailable.

However, you can also specify that you want your columns to be nested below the table name like this:

```
var options = {sql: '...', nestTables: true};
connection.query(options, function (error, results, fields) {
  if (error) throw error;
  /* results will be an array like this now:
  [{
    table1: {
      fieldA: '...',
      fieldB: '...',
    },
    table2: {
      fieldA: '...',
      fieldB: '...',
    },
  }, ...]
  */
});
```

Or use a string separator to have your results merged.

```
var options = {sql: '...', nestTables: '_'};
connection.query(options, function (error, results, fields) {
  if (error) throw error;
  /* results will be an array like this now:
  [{
```

```
    table1_fieldA: '...',
    table1_fieldB: '...',
    table2_fieldA: '...',
    table2_fieldB: '...',
  }, ...]
  */
});
```

## Transactions

Simple transaction support is available at the connection level:

```
connection.beginTransaction(function(err) {
  if (err) { throw err; }
  connection.query('INSERT INTO posts SET title=?', title, function (e
    if (error) {
      return connection.rollback(function() {
        throw error;
      });
    }

    var log = 'Post ' + results.insertId + ' added';

    connection.query('INSERT INTO log SET data=?', log, function (erro
      if (error) {
        return connection.rollback(function() {
          throw error;
        });
      }
      connection.commit(function(err) {
        if (err) {
          return connection.rollback(function() {
            throw err;
```

```
          });
        }
        console.log('success!');
      });
    });
  });
});
```

Please note that beginTransaction(), commit() and rollback() are simply convenience functions that execute the START TRANSACTION, COMMIT, and ROLLBACK commands respectively. It is important to understand that many commands in MySQL can cause an implicit commit, as described **in the MySQL documentation**

# Ping

A ping packet can be sent over a connection using the `connection.ping` method. This method will send a ping packet to the server and when the server responds, the callback will fire. If an error occurred, the callback will fire with an error argument.

```
connection.ping(function (err) {
  if (err) throw err;
  console.log('Server responded to ping');
})
```

# Timeouts

Every operation takes an optional inactivity timeout option. This allows you to specify appropriate timeouts for operations. It is important to note that these timeouts are not part of the MySQL protocol, and rather timeout operations through the client. This means that when a timeout is reached, the connection it occurred on will be destroyed and no further operations can be performed.

```
// Kill query after 60s
connection.query({sql: 'SELECT COUNT(*) AS count FROM big_table', time
  if (error && error.code === 'PROTOCOL_SEQUENCE_TIMEOUT') {
    throw new Error('too long to count table rows!');
  }

  if (error) {
    throw error;
  }

  console.log(results[0].count + ' rows');
});
```

# Error handling

This module comes with a consistent approach to error handling that you should review carefully in order to write solid applications.

Most errors created by this module are instances of the JavaScript **Error** object. Additionally they typically come with two extra properties:

- `err.code` : String, contains the MySQL server error symbol if the error is a MySQL server error (e.g. `'ER_ACCESS_DENIED_ERROR'`), a Node.js error code if it is a Node.js error (e.g. `'ECONNREFUSED'`), or an internal error code (e.g. `'PROTOCOL_CONNECTION_LOST'`).
- `err.errno` : Number, contains the MySQL server error number. Only populated from MySQL server error.
- `err.fatal` : Boolean, indicating if this error is terminal to the connection object. If the error is not from a MySQL protocol operation, this property will not be defined.
- `err.sql` : String, contains the full SQL of the failed query. This can be useful when using a higher level interface like an ORM that is generating the queries.
- `err.sqlState` : String, contains the five-character SQLSTATE value. Only populated from MySQL server error.

- `err.sqlMessage` : String, contains the message string that provides a textual description of the error. Only populated from MySQL server error.

Fatal errors are propagated to *all* pending callbacks. In the example below, a fatal error is triggered by trying to connect to an invalid port. Therefore the error object is propagated to both pending callbacks:

```
var connection = require('mysql').createConnection({
  port: 84943, // WRONG PORT
});

connection.connect(function(err) {
  console.log(err.code); // 'ECONNREFUSED'
  console.log(err.fatal); // true
});

connection.query('SELECT 1', function (error, results, fields) {
  console.log(error.code); // 'ECONNREFUSED'
  console.log(error.fatal); // true
});
```

Normal errors however are only delegated to the callback they belong to. So in the example below, only the first callback receives an error, the second query works as expected:

```
connection.query('USE name_of_db_that_does_not_exist', function (error
  console.log(error.code); // 'ER_BAD_DB_ERROR'
});

connection.query('SELECT 1', function (error, results, fields) {
  console.log(error); // null
  console.log(results.length); // 1
});
```

Last but not least: If a fatal errors occurs and there are no pending callbacks, or a normal error occurs which has no callback belonging to it, the error is emitted as an `'error'` event on the connection object. This is demonstrated in the example below:

```
connection.on('error', function(err) {
  console.log(err.code); // 'ER_BAD_DB_ERROR'
});

connection.query('USE name_of_db_that_does_not_exist');
```

Note: `'error'` events are special in node. If they occur without an attached listener, a stack trace is printed and your process is killed.

**tl;dr:** This module does not want you to deal with silent failures. You should always provide callbacks to your method calls. If you want to ignore this advice and suppress unhandled errors, you can do this:

```
// I am Chuck Norris:
connection.on('error', function() {});
```

## Exception Safety

This module is exception safe. That means you can continue to use it, even if one of your callback functions throws an error which you're catching using 'uncaughtException' or a domain.

## Type casting

For your convenience, this driver will cast mysql types into native JavaScript types by default. The following mappings exist:

### Number

- TINYINT

- SMALLINT
- INT
- MEDIUMINT
- YEAR
- FLOAT
- DOUBLE

### Date

- TIMESTAMP
- DATE
- DATETIME

### Buffer

- TINYBLOB
- MEDIUMBLOB
- LONGBLOB
- BLOB
- BINARY
- VARBINARY
- BIT (last byte will be filled with 0 bits as necessary)

### String

**Note** text in the binary character set is returned as `Buffer`, rather than a string.

- CHAR
- VARCHAR
- TINYTEXT
- MEDIUMTEXT
- LONGTEXT
- TEXT
- ENUM
- SET
- DECIMAL (may exceed float precision)
- BIGINT (may exceed float precision)

- TIME (could be mapped to Date, but what date would be set?)
- GEOMETRY (never used those, get in touch if you do)

It is not recommended (and may go away / change in the future) to disable type casting, but you can currently do so on either the connection:

```
var connection = require('mysql').createConnection({typeCast: false});
```

Or on the query level:

```
var options = {sql: '...', typeCast: false};
var query = connection.query(options, function (error, results, fields
  if (error) throw error;
  // ...
});
```

### Custom type casting

You can also pass a function and handle type casting yourself. You're given some column information like database, table and name and also type and length. If you just want to apply a custom type casting to a specific type you can do it and then fallback to the default.

The function is provided two arguments `field` and `next` and is expected to return the value for the given field by invoking the parser functions through the `field` object.

The `field` argument is a `Field` object and contains data about the field that need to be parsed. The following are some of the properties on a `Field` object:

- `db` - a string of the database the field came from.
- `table` - a string of the table the field came from.
- `name` - a string of the field name.
- `type` - a string of the field type in all caps.
- `length` - a number of the field length, as given by the database.

The `next` argument is a `function` that, when called, will return the default type conversion for the given field.

When getting the field data, the following helper methods are present on the `field` object:

- `.string()` - parse the field into a string.
- `.buffer()` - parse the field into a `Buffer`.
- `.geometry()` - parse the field as a geometry value.

The MySQL protocol is a text-based protocol. This means that over the wire, all field types are represented as a string, which is why only string-like functions are available on the `field` object. Based on the type information (like `INT`), the type cast should convert the string field into a different JavaScript type (like a `number`).

Here's an example of converting `TINYINT(1)` to boolean:

```
connection = mysql.createConnection({
  typeCast: function (field, next) {
    if (field.type === 'TINY' && field.length === 1) {
      return (field.string() === '1'); // 1 = true, 0 = false
    } else {
      return next();
    }
  }
});
```

**WARNING: YOU MUST INVOKE the parser using one of these three field functions in your custom typeCast callback. They can only be called once.**

# Debugging and reporting problems

If you are running into problems, one thing that may help is enabling the `debug` mode for the connection:

```
var connection = mysql.createConnection({debug: true});
```

This will print all incoming and outgoing packets on stdout. You can also restrict debugging to packet types by passing an array of types to debug:

```
var connection = mysql.createConnection({debug: ['ComQueryPacket', 'Rc
```

to restrict debugging to the query and data packets.

If that does not help, feel free to open a GitHub issue. A good GitHub issue will have:

- The minimal amount of code required to reproduce the problem (if possible)
- As much debugging output and information about your environment (mysql version, node version, os, etc.) as you can gather.

# Security issues

Security issues should not be first reported through GitHub or another public forum, but kept private in order for the collaborators to assess the report and either (a) devise a fix and plan a release date or (b) assert that it is not a security issue (in which case it can be posted in a public forum, like a GitHub issue).

The primary private forum is email, either by emailing the module's author or opening a GitHub issue simply asking to whom a security issues should be addressed to without disclosing the issue or type of issue.

An ideal report would include a clear indication of what the security issue is and how it would be exploited, ideally with an accompanying proof of concept ("PoC") for collaborators to work against and validate potentional fixes against.

# Contributing

This project welcomes contributions from the community. Contributions are accepted using GitHub pull requests. If you're not familiar with making GitHub pull requests, please refer to the **GitHub documentation "Creating a pull request"**.

For a good pull request, we ask you provide the following:

1. Try to include a clear description of your pull request in the description. It should include the basic "what" and "why"s for the request.
2. The tests should pass as best as you can. See the **Running tests** section on how to run the different tests. GitHub will automatically run the tests as well, to act as a safety net.
3. The pull request should include tests for the change. A new feature should have tests for the new feature and bug fixes should include a test that fails without the corresponding code change and passes after they are applied. The command `npm run test-cov` will generate a `coverage/` folder that contains HTML pages of the code coverage, to better understand if everything you're adding is being tested.
4. If the pull request is a new feature, please be sure to include all appropriate documentation additions in the `Readme.md` file as well.
5. To help ensure that your code is similar in style to the existing code, run the command `npm run lint` and fix any displayed issues.

# Running tests

The test suite is split into two parts: unit tests and integration tests. The unit tests run on any machine while the integration tests require a MySQL server instance to be setup.

### Running unit tests

```
$ FILTER=unit npm test
```

### Running integration tests

Set the environment variables `MYSQL_DATABASE`, `MYSQL_HOST`, `MYSQL_PORT`, `MYSQL_USER` and `MYSQL_PASSWORD`. `MYSQL_SOCKET` can also be used in place of `MYSQL_HOST` and `MYSQL_PORT` to connect over a UNIX socket. Then run `npm test`.

For example, if you have an installation of mysql running on localhost:3306 and no password set for the `root` user, run:

```
$ mysql -u root -e "CREATE DATABASE IF NOT EXISTS node_mysql_test"
$ MYSQL_HOST=localhost MYSQL_PORT=3306 MYSQL_DATABASE=node_mysql_test
```

# Todo

- Prepared statements
- Support for encodings other than UTF-8 / ASCII

## Keywords

none

### Install

```
> npm i mysql
```

### Repository

github.com/mysqljs/mysql

### Homepage

github.com/mysqljs/mysql#readme

### Weekly Downloads

831,261

### Version

2.18.1

### License

MIT

### Unpacked Size

482 kB

### Total Files

61

### Issues

173

### Pull Requests

42

### Last publish

2 years ago

## Collaborators



>_Try on RunKit

Report malware

### Support

Help

Advisories

Status

Contact npm

### Company

About

Blog

Press

## Terms & Policies

Policies

Terms of Use

Code of Conduct

Privacy