How To Use MySQL With Node & Express

by Stefan Fidanov

Everybody is talking about NoSQL, especially in the NodeJS world. Lot's of people even associate Node with Mongo and other NoSQL databases.

However, the world doesn't end there. SQL databases like MySQL, PostgreSQL, Oracle or even SQL Server are battle tested in all kind of scenarios. Companies both large and small use them to run their mission critical systems.

Moreover, if you are already comfortable with MySQL, Postrgres or any other there is not reason to switch to another database.

Often, it's better to use what you know and what you love, instead of changing to another technology for promises that might never materialize.

Actually, even though the last few years people talk mostly about NoSQL, most of the web apps that you encounter today are still running on SQL databases. The most popular of which is still MySQL.

Unfortunately, many people take for granted that Node should be used with a NoSQL database and that is why we are not hearing what can be done with SQL.

Let's have a look what we can do with MySQL.

Connecting to MySQL

```
$ npm install mysql
```

mysql is a great module which makes working with MySQL very easy and it provides all the capabilities you might need.

Once you have **mysql** installed, all you have to do to connect to your database is

```
var mysql = require('mysql')

var connection = mysql.createConnection({
  host: 'localhost',
  user: 'your_user',
  password: 'some_secret',
  database: 'the_app_database'
})

connection.connect(function(err) {
  if (err) throw err
  console.log('You are now connected...')
})
```

Now you can begin writing and reading from your database.

Reading and Writing to MySQL

You know how you can connect, so let's have a look at a simple example

```
var mysql = require('mysql')

var connection = mysql.createConnection({
  host: 'localhost',
  user: 'your_user',
  password: 'some_secret',
```

```
connection.connect(function(err) {
  if (err) throw err
  console.log('You are now connected...')

  connection.query('CREATE TABLE people(id int primary key, name varchar(255),
    if (err) throw err
    connection.query('INSERT INTO people (name, age, address) VALUES (?, ?, ?)
      if (err) throw err
      connection.query('SELECT * FROM people', function(err, results) {
        if (err) throw err
        console.log(results[0].id)
        console.log(results[0].name)
        console.log(results[0].age)
        console.log(results[0].address)
      })
    })
  })
})
```

First, you connect to the database, then you insert one record and then you read it back.

You can also see that **?** acts as placeholders for your values. It not only makes using values easier but it also escapes them so that your queries are always safe.

Replacing your DB file for help

As you saw using the **mysql** module is very easy, but real web apps have more complex needs. That is why in Connecting and Working with MongoDB we created a separate file **db.js** to help us manage our connections.

Let's look how your helper **db.js** file will look like when using MySQL instead of Mongo. Its purpose is to have easy access to the database whenever you need it without constantly entering credentials.

```
var mysql = require('mysql')
  , async = require('async')

var PRODUCTION_DB = 'app_prod_database'
  , TEST_DB = 'app_test_database'

exports.MODE_TEST = 'mode_test'
exports.MODE_PRODUCTION = 'mode_production'

var state = {
  pool: null,
  mode: null,
}

exports.connect = function(mode, done) {
  state.pool = mysql.createPool({
    host: 'localhost',
    user: 'your_user',
    password: 'some_secret',
    database: mode === exports.MODE_PRODUCTION ? PRODUCTION_DB : TEST_DB
  })

  state.mode = mode
  done()
}

exports.get = function() {
  return state.pool
}

exports.fixtures = function(data) {
  var pool = state.pool
  if (!pool) return done(new Error('Missing database connection.'))

  var names = Object.keys(data.tables)
  async.each(names, function(name, cb) {
    async.each(data.tables[name], function(row, cb) {
```

Close

```
      pool.query('INSERT INTO ' + name + ' (' + keys.join(',') + ') VALUES ('
    }, cb)
  }, done)
}

exports.drop = function(tables, done) {
  var pool = state.pool
  if (!pool) return done(new Error('Missing database connection.'))

  async.each(tables, function(name, cb) {
    pool.query('DELETE * FROM ' + name, cb)
  }, done)
}
```

This **db.js** file is a little bit more complicated than what we did before.

First, it provides a way to connect to the database. When you connect you can do it either in production mode or in test mode. Test mode is for only when running automated tests.

Then there is a get method which can always provide you with an active connection, which you can use to query the database.

So whenever you need to contact the database instead of setting up database passwords and other arguments you just call this method and you are ready to go.

Finally, there are two more methods **fixtures** and **drop**, which exist to make your life easier when testing.

drop clears the data, but not the schemas from all the tables that you want. It will help you to be sure that your test database is always clean before every test.

fixtures takes a JSON object and loads its data into the database, so that there is something on which to run your tests. Let's have a quick look how it looks to work with it:

```
var data = {
  tables: {
    people: [
     {id: 1, name: "John", age: 32},
     {id: 2, name: "Peter", age: 29},
    ],
    cars: [
      {id: 1, brand: "Jeep", model: "Cherokee", owner_id: 2},
      {id: 2, brand: "BMW", model: "X5", owner_id: 2},
      {id: 3, brand: "Volkswagen", model: "Polo", owner_id: 1},
    ],
  },
}

var db = require('./db')
db.connect(db.MODE_PRODUCTION, function() {
  db.fixtures(data, function(err) {
    if (err) return console.log(err)
    console.log('Data has been loaded...')
  })
})
```

It is very simple to use, and after running it your tables *cars* and *people* will have data in them.

Building models with SQL

Everything is in place and the next step is to actually see how you are going to use **db.js** so that it will make your life easier.

Let's have an example app with the following structure

```
controllers/
  comments.js
  users.js
```

```
  user.js
views/
app.js
db.js
package.json
```

app.js is the entrypoint of the application and this is the place where we are going to setup the database connection. Let's have a look what is inside it.

```
var db = require('./db')

app.use('/comments', require('./controllers/comments'))
app.use('/users', require('./controllers/users'))

// Connect to MySQL on start
db.connect(db.MODE_PRODUCTION, function(err) {
  if (err) {
    console.log('Unable to connect to MySQL.')
    process.exit(1)
  } else {
    app.listen(3000, function() {
      console.log('Listening on port 3000...')
    })
  }
})
```

Your app will interact with the database through its models. So let's have a look how a model can look like. For example this is how your comments model can look like:

```
var db = require('../db.js')

exports.create = function(userId, text, done) {
  var values = [userId, text, new Date().toISOString()]
```

```
    done(null, result.insertId)
  })
}

exports.getAll = function(done) {
  db.get().query('SELECT * FROM comments', function (err, rows) {
    if (err) return done(err)
    done(null, rows)
  })
}

exports.getAllByUser = function(userId, done) {
  db.get().query('SELECT * FROM comments WHERE user_id = ?', userId, function
    if (err) return done(err)
    done(null, rows)
  })
}
```

As you can see the **db.js** files makes it very easy to build any kind of models without worrying about connecting the database. It doesn't even know whether you are in production or testing mode, so your models will work in both cases.

Then you can use your newly built models in your controllers and they won't even know that there is a SQL solution behind.

Advanced usage: using different database instance for reading and writing

As your app grow your needs grow. Many of todays applications are read orientied.

That means that people write data more rarely than they read. For example, all social networks are this type of applications. Usually a status is written once but then read by hundreds or even thousands of people.

Then when you have more users you can easily add more database only for reading and your app will scale and still be super fast.

But how can this work in our setup from above?

The **mysql** module has the very useful PoolCluster feature, which can take the configurations for several instances and then connect to all of them. Let's see how your **db.js** file will look:

```
var mysql = require('mysql')
  , async = require('async')

var PRODUCTION_DB = 'app_prod_database'
  , TEST_DB = 'app_test_database'

exports.MODE_TEST = 'mode_test'
exports.MODE_PRODUCTION = 'mode_production'

var state = {
  pool: null,
  mode: null,
}

exports.connect = function(mode, done) {
  if (mode === exports.MODE_PRODUCTION) {
    state.pool = mysql.createPoolCluster()

    state.pool.add('WRITE', {
      host: '192.168.0.5',
      user: 'your_user',
      password: 'some_secret',
      database: PRODUCTION_DB
    })

    state.pool.add('READ1', {
      host: '192.168.0.6',
```

```
    })

    state.pool.add('READ2', {
      host: '192.168.0.7',
      user: 'your_user',
      password: 'some_secret',
      database: PRODUCTION_DB
    })
  } else {
    state.pool = mysql.createPool({
      host: 'localhost',
      user: 'your_user',
      password: 'some_secret',
      database: TEST_DB
    })
  }

  state.mode = mode
  done()
}

exports.READ = 'read'
exports.WRITE = 'write'

exports.get = function(type, done) {
  var pool = state.pool
  if (!pool) return done(new Error('Missing database connection.'))

  if (type === exports.WRITE) {
    state.pool.getConnection('WRITE', function (err, connection) {
      if (err) return done(err)
      done(null, connection)
    })
  } else {
    state.pool.getConnection('READ*', function (err, connection) {
      if (err) return done(err)
      done(null, connection)
    })
```

Close

```
exports.fixtures = function(data) {
  var pool = state.pool
  if (!pool) return done(new Error('Missing database connection.'))

  var names = Object.keys(data.tables)
  async.each(names, function(name, cb) {
    async.each(data.tables[name], function(row, cb) {
      var keys = Object.keys(row)
        , values = keys.map(function(key) { return "'" + row[key] + "'" })

      pool.query('INSERT INTO ' + name + ' (' + keys.join(',') + ') VALUES ('
    }, cb)
  }, done)
}

exports.drop = function(tables, done) {
  var pool = state.pool
  if (!pool) return done(new Error('Missing database connection.'))

  async.each(tables, function(name, cb) {
    pool.query('DELETE * FROM ' + name, cb)
  }, done)
}
```

There are three main differences in what you had before.

When connecting in the `connection` method in test mode, you what you did before. However, when you are in production mode, you add 3 more servers.

Each server has a name. The first is WRITE and then there are READ1 and READ2.

It should be pretty clear, what the purpose of each one of them is. READ1 and READ2 are slaves of WRITE, so everytime someone write data to WRITE it will be shortly afterwards available to READ1 and READ2.

Close

argument.

The last change is again on the `get` method. When the you want a database connection you need to tell what type of connection you want: READ or WRITE.

Then based on the names set for the servers the correct type of connection will be provided. You can see that `getConnection` in this case provides a type of pattern mathching so that you can select the appropriate database servers.

This is all you need and now you can use your db file and you can scale as much as you want.

Let's have a quick look how the model from above will change

```
var db = require('../db.js')

exports.create = function(userId, text, done) {
  var values = [userId, text, new Date().toISOString()]

  db.get(db.WRITE, function(err, connection) {
    if (err) return done('Database problem')

    connection.query('INSERT INTO comments (user_id, text, date) VALUES(?, ?,
      if (err) return done(err)
      done(null, result.insertId)
    })
  })
}

exports.getAll = function(done) {
  db.get(db.READ, function(err, connection) {
    if (err) return done('Database problem')

    connection.query('SELECT * FROM comments', function (err, rows) {
      if (err) return done(err)
      done(null, rows)
```

Close

```
exports.getAllByUser = function(userId, done) {
  db.get(db.READ, function(err, connection) {
    if (err) return done('Database problem')

    connection.query('SELECT * FROM comments WHERE user_id = ?', userId, funct
      if (err) return done(err)
      done(null, rows)
    })
  })
}
```

In each of the methods you select the appropriate connection, which reads from a purposely build database.

This can be even improved. You can create a proxy connection object, which based on the query can understand whether this is a write or read query and then select the appropriate connection. This way you won't even change your models.

Schemas & Migrations

Up until know I didn't talk about how you load the database schema, but you need one or none of the code from above will run.

Basically, there are two ways to load the schema. You can either run it manually from an interface like phpMyAdmin (I know it is php :-) or you can try to automate it.

I personally prefer to do it manually, but I always keep an exported version of the entire schema.

Next

For your next project don't go automatically to Mongo or other NoSQL databases

Other articles that you may like

- Hosting & Deploying NodeJS Apps on Ubuntu
- Hosting & Deploying NodeJS Apps on CentOS

Did you like this article?

Please share it on

We have created a video course for you to teach you what we have learned from years of development with Node & Express.

It is all about best practices which make development and maintenance of Express apps simple again.

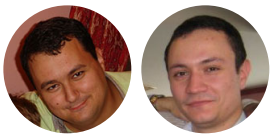
Join our video course on best practices in ExpressJS

Freelancing is hard. You feel alone. You have to manage everything yourself and some clients make it even harder than it should be.

Never ending projects. Clients constantly calling you for no reason.

Sounds familiar? Join hundreds of freelancers with our free

7-part guide to dealing with difficult clients



We are Stefan Fidanov & Vasil Lyutskanov. We share actionable advice about development with Node, Express, React and other web & mobile technologies.

It is everything that we have learned from years of experience working with customers from all over the world on projects of all sizes.

Follow @fidanov

Close