Installing

Before we begin you need to go to www.mongodb.org to download and run the latest version of Mongo. Follow the instructions and you will be ready in less than 5 minutes.

Once you have the database up and running, you will have to install the official Mongo driver to connect to it. It is a NPM package which provides useful methods for working with Mongo from Express and Node.

```
$ npm install mongodb
```

Now you are ready to begin.

Connecting

Let's look at our first example.

```
var MongoClient = require('mongodb').MongoClient

var URL = 'mongodb://localhost:27017/mydatabase'

MongoClient.connect(URL, function(err, db) {
  if (err) return

  var collection = db.collection('foods')
  collection.insert({name: 'taco', tasty: true}, function(err,
result) {
    collection.find({name: 'taco'}).toArray(function(err, docs)
{
      console.log(docs[0])
      db.close()
    })
  })
})
```

This is a very simple example. It connects to Mongo, then it saves some data (`{name:'taco', tasty: true}`) inside the `foods` collection and then it reads and displays the same data.

To run the code from above, you can put it in `index.js` and then do the following.

```
$ node index.js
```

If Mongo is running and you have installed the NPM package you should see as a result something like the line below.

```
{ "_id": 551e56b41143e39e7ff6b272, "name": "taco", "tasty": true
}
```

If you run this file multiple times it will create many of the same objects inside the collection but with different **_id** each time.

For clarity, the example above has been kept intentionally simple. I've omitted all kind of error checks, like for example I don't check whether the insert was a successful or even if reading the value produced a result.

Reusing the connection

Let's go a little bit further. Let's assume we have an app with the following structure

```
app/
  controllers/
    comments.js
    users.js
  models/
  views/
  app.js
```

For this example we will need to also install two more NPM packages.

```
$ npm install express jade
```

Express will serve our web app and jade will render its templates.

In file structure above, there are two controllers **comments** and **users**. The comments controller displays and edits comments, where as the other one does the same for users.

Both of them need to connect to the database. If you just copy the code from the previous example, inside each of these files, it will work but it won't be very convenient.

When you later want to make a change to how you connect to Mongo, you will have to modify each file. The more files you have the more difficult and error prone it will be.

Instead, let's create in our app folder a **db.js** file with the following content. It will help us manage our database connections and more.

```
var MongoClient = require('mongodb').MongoClient

var state = {
  db: null,
}

exports.connect = function(url, done) {
  if (state.db) return done()

  MongoClient.connect(url, function(err, db) {
    if (err) return done(err)
    state.db = db
    done()
  })
}

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

exports.close = function(done) {
  if (state.db) {
    state.db.close(function(err, result) {
      state.db = null
      state.mode = null
```

```
      done(err)
    })
  }
}
```

This simple file will help us connect to the database when the app starting and then any controller can just use the the **db** object returned by the **get** method.

It will always be the same **db** object because **require** caches the result the first time it is called. Therefore it will return the same object, which will have the same **get** method, which in return will have access to the same **state.db** variable.

Let's see how our **app.js** file will look.

```
var express = require('express')
  , app = express()

var db = require('./db')

app.engine('jade', require('jade').__express)
app.set('view engine', 'jade')

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

// Connect to Mongo on start
db.connect('mongodb://localhost:27017/mydatabase', function(err)
{
  if (err) {
    console.log('Unable to connect to Mongo.')
    process.exit(1)
  } else {
    app.listen(3000, function() {
      console.log('Listening on port 3000...')
    })
  }
})
```

This is the entry point of our application. It configures the app and then connects to the database.

Once the application is running it connects to Mongo so that other components can use the already established connections with the database.

For example if we want to display some comments, we can provide the following paths in the **comments.js** controller:

```
var express = require('express')
  , router = express.Router()

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

router.get('/all', function(req, res) {
  var collection = db.get().collection('comments')

  collection.find().toArray(function(err, docs) {
    res.render('comments', {comments: docs})
  })
})

router.get('/recent', function(req, res) {
  var collection = db.get().collection('comments')

  collection.find().sort({'date': -
1}).limit(100).toArray(function(err, docs) {
    res.render('comments', {comments: docs})
  })
})

module.exports = router
```

The file above provides two paths. The first one display all comments, where as the second one displays the hundred most recent comments sorted by date.

The other controller for the users can also reuse the database connection just as easily.

Just in the first example, this example is kept intentionally simple. There are few checks for errors. Also our **db.js** file contains just the bare functionality.

In a more complex application it can provide many more useful methods including some which will help test the application.

Building models with business logic

Reusing the connection for different controllers is a good thing, but we can further improve how we work with Mongo. For example, you may need to get all comments for many different tasks like displaying them or running some statistics on them or do something else.

With our current setup this will require that you copy the following code each time:

```
collection.find().toArray(function(err, docs) {
  // Do something...
})
```

Again, when you later want to make a change, you will have to make the change everywhere. It is the same problem as earlier.

But this is not the only one. The way it works now, requires that you controllers know database specific details.

To improve this lets go full Model-View-Controller pattern and implement some models.

```
app/
  controllers/
    comments.js
    users.js
  models/
    comments.js
    users.js
  views/
  app.js
```

As you can see we will have two more files in the models folder with content looking like the following

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

exports.all = function(cb) {
  var collection = db.get().collection('comments')

  collection.find().toArray(function(err, docs) {
    cb(err, docs)
  })
}

exports.recent = function(cb) {
  var collection = db.get().collection('comments')

  collection.find().sort({'date': -
1}).limit(100).toArray(function(err, docs) {
    cb(err, docs)
  })
}
```

Our comments model file uses the established database connection to request the data.

To use the model you will have to change a little bit in the controllers.

```
var express = require('express')
  , router = express.Router()

var Comments = require('../models/comments')

router.get('/all', function(req, res) {
  Comments.all(function(err, docs) {
    res.render('comments', {comments: docs})
  })
})

router.get('/recent', function(req, res) {
  Comments.recent(function(err, docs) {
    res.render('comments', {comments: docs})
  })
})

module.exports = router
```

At the end our models share the database management file **db.js** and they are the only ones who are aware how we are requesting our data from Mongo.

Xây dựng module db CSDL MySQL

Connecting to MySQL

Before you do anything, you need to install the right NPM package.

```
$ 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',
  database: 'the_app_database'
})

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), age int, address text)', function(err, result) {
    if (err) throw err
    connection.query('INSERT INTO people (name, age, address)
VALUES (?, ?, ?)', ['Larry', '41', 'California, USA'],
function(err, result) {
      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.

Its second goal is to make it easy to run tests which access your database.

```
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) {
      var keys = Object.keys(row)
        , values = keys.map(function(key) { return "'" +
row[key] + "'" })

      pool.query('INSERT INTO ' + name + ' (' + keys.join(',') +
') VALUES (' + values.join(',') + ')', cb)
    }, 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
models/
  comment.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()]
```

```
  db.get().query('INSERT INTO comments (user_id, text, date)
VALUES(?, ?, ?)', values, function(err, result) {
    if (err) return done(err)
    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 (err, rows) {
    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.

In this kind of situations it makes sense to have one main database which will accept all the writes and then synced to a few more MySQL instances which will serve only the read requests.

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',
      user: 'your_user',
      password: 'some_secret',
      database: PRODUCTION_DB
```

```
    })

    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)
    })
  }
}

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 (' + values.join(',') + ')', cb)
    }, 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.

The second change is the `get` method. Before, it returned a connection immediately, right now it returns a connection to the callback provided as a second 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(?, ?, ?)', values, function(err, result) {
      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)
    })
  })
}

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, function (err, rows) {
      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.