## Prerequisites

A node project with **mysql** used for database.

## Install

It can be installed using npm.

npm install mysql-migrations

**Setup**

1. Create a directory where you wish to maintain all your migrations. We call it migrations.
2. Instantiate mysql-migrations by passing a mysql pool and the migrations directory path.



# Database Configurations

Now config.json file which is located at config\config.json and update your database details,



# Generating Model & Migrations

Now again we gonna use sequelize cli command to generate model and migrations files.

We are going to have two models, Company and Employee.

The relationship between a Company it's Employee is going to be one-to-many, such that a Company can have many Employees while a Employee can only belong to one Company .

# migration.js

var mysql = require('mysql');

var migration = require('mysql-migration');

var connection = mysql.createPool({

connectionLimit : 10,

host : 'localhost',

user : 'root',

password : 'password',

database : 'your\_database'

});

migration.init(connection, \_\_dirname + '/migrations');

Database migration is the process of transforming data between various states without any human interaction. This process will allow us to track changes between schema updates.

**Installation**

The node-db-migrate framework includes a neat CLI tool we can use to manage our database migration scripts. We can install it globally on our machine using the following command:

npm install -g db-migrate

Make sure you are in the working directory of your Node.js application.

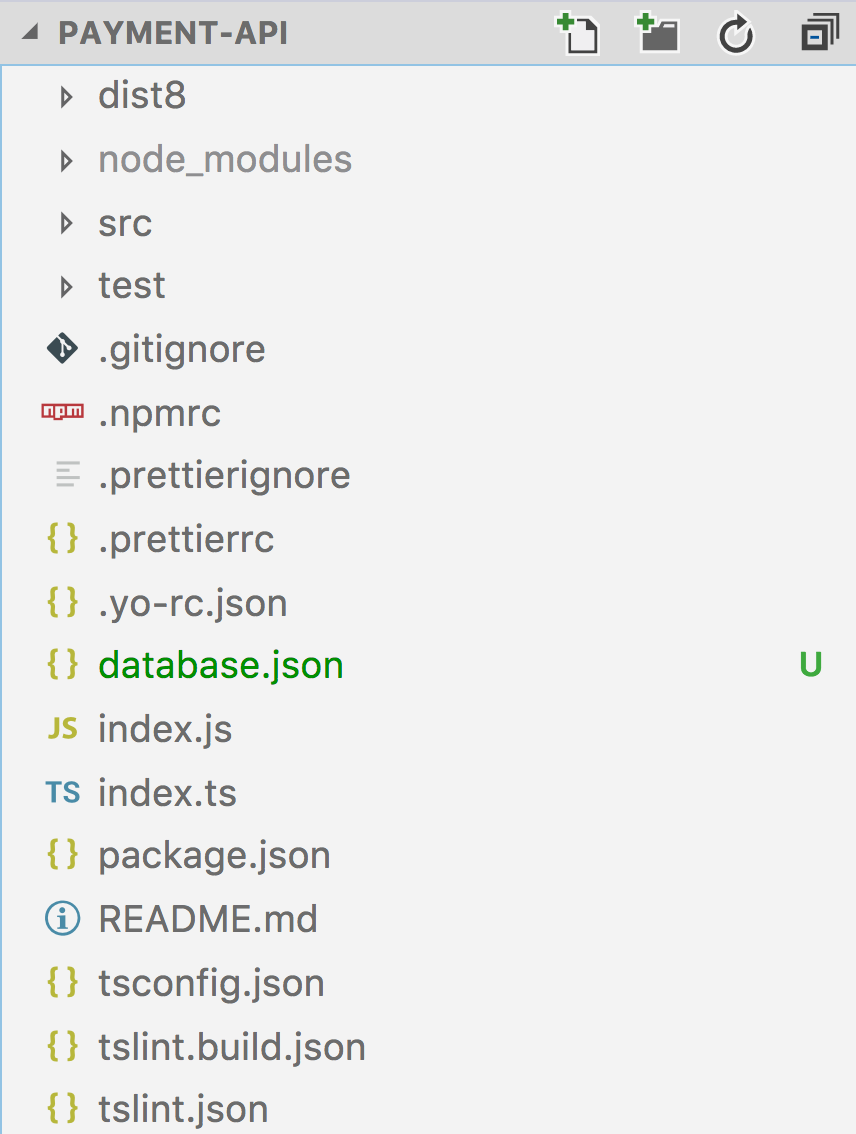
We will be managing a MySQL database schema. node-db-migrate has various packages that take care of connecting to the various database engines, such as MySQL. In order to manage a MySQL database using node-db-migrate, we need to install the following package in our project:

npm install --save db-migrate-mysql

**Configuration**

We need to tell the node-db-migrate framework how to connect to our database. I have installed a local MySQL server and created a schema called “payment\_api”.

**Schema**: The definition that describes the entire collection of objects in a database. We refer to this as the database on a MySQL server.**Collation**: A collation is a set of rules for comparing characters in a character set. We are using the utf8 default collation in order to support a wide range of character sets in our application.



{  
"dev": {  
"driver": "mysql",  
"host": "localhost",  
"port": "3306",  
"user": "root",  
"password": "root",  
"database": "payment\_api"  
}

}

The database.json file contains a single JSON object with a single property called **“dev”**

The configuration properties are pretty self-explanatory. The only one worth mentioning is the **“driver”** property. We are using **“mysql”** as the property value, which will indicate to the node-db-migrate framework that it needs to use the db-migrate-mysql npm package we installed earlier to connect to the database.

## Creating a new Migration Script

We can create a new migration script by running the following command:

db-migrate create user[INFO] Created migration at /projects/payment-api/migrations/20180606161337-user.js

## Exploring the Migration Script

Every migration framework usually has two functions we can use to implement our migration up and down.

Up: Contains code to move the database forward, or up, to the next database version.  
Down: Contains code to move the database backwards, or down, to the previous database version i.e. **rollback!**

exports.up = function(db, callback) {  
 return null;  
};exports.down = function(db, callback) {  
 return null;  
};

Both functions pass in an already connected db instance, which is taken care of through the database.json configuration file used by the node-db-migrate framework. The functions also pass in a second callback parameter, which is not immediately clear after generating the script (and only apparent after studying the documentation). I have included it in the above snippet. It is important to include the callback parameter as our migration scripts are asynchronous. The code needs to callback to the node-db-migrate framework after completing database operations.

We will be using the db#createTable and db#dropTable functions.

exports.up = function(db, callback) {

db.createTable('user', {  
 id: {  
 type: 'int',  
 primaryKey: true  
 },  
 full\_name: {  
 type: 'string',  
 length: 40  
 },  
 dob: {  
 type: 'date'  
 },  
 email: {  
 type: 'string',  
 length: 50  
 },  
 }, function(err) {  
 if (err) return callback(err);  
 return callback();  
 });};exports.down = function(db, callback) { db.dropTable('user', callback);};

Our code should create a new table when we move the database version forward and drop the table when reverting back. Lets test this out.

db-migrate up[INFO] Processed migration 20180606161337-user  
[INFO] Done

Making a change is simple:

db-migrate create update-user[INFO] Created migration at /payment-api/migrations/20180606164321-update-user.js

We want to apply the following change:

1. Add a new column called *firstname*
2. Add a new column called *lastname*
3. Move all of the existing data from the *full\_name* column into the separate *firstname*and *lastname* columns *(This will be explored in more detail in a future article, for now we will focus on just the schema definition updates).*
4. Remove the old column called *full\_name*

exports.up = function (db, callback) { db.addColumn('user', 'firstname', {  
 type: 'string',  
 length: 50  
 }, function(err) {  
 if (err) return callback(err);  
   
 db.addColumn('user', 'lastname', {  
 type: 'string',  
 length: 50  
 }, function(err) {  
 if (err) return callback(err); db.removeColumn('user', 'full\_name', callback);  
 });  
 });};exports.down = function (db, callback) { db.addColumn('user', 'full\_name', {  
 type: 'string',  
 length: 50  
 }, function(err) {  
 if (err) return callback(err);  
   
 db.removeColumn('user', 'lastname', function(err) {  
 if (err) return callback(err);  
 db.removeColumn('user', 'firstname', callback)  
 });  
 });};

*We are skipping the actual data migration from****full\_name****to****firstname****and****lastname****for the sake of simplicity. This will be explored in a later article.*

db-migrate up

## Adding Migrations

### Initiate a migration

Run node migration.js add migration create\_table\_users. Now open the migrations folder. Locate the newest file with greatest timestamp as it predecessor. The file will have the name which was specified in the command such as 12213545345\_create\_table\_users.js

### Add migrations

Write the query in up key of the json created for the forward migration. As a part of good practice, also write the script to rollback the migration in down key. Ex.

module.exports = {

"up": "CREATE TABLE users (user\_id INT NOT NULL, UNIQUE KEY user\_id (user\_id), name TEXT )",

"down": "DROP TABLE users"

}

### Add seed

Run node migration.js add seed create\_table\_users to add a seed.

module.exports = {

"up": "UPDATE users SET name = 'John Snow' WHERE name = ''",

"down": "UPDATE users SET name = '' WHERE name = 'John Snow'"

}

### Initate and Add migration in single command

Run node migration.js add migration create\_table\_users "CREATE TABLE mysql\_migrations\_347ertt3e (user\_id INT NOT NULL, UNIQUE KEY user\_id (user\_id) )". Locate the newest file with greatest timestamp as it predecessor and open it. Query will be automatically added as up key. However down key needs to be filled manually.

### Custom migrations

You may initiate the migration file and add a function.

module.exports = {

'up' : function (conn, cb) {

conn.query ("UPDATE users set name = 'alen'", function (err, res) {

cb();

});

},

'down' : ""

}

## Executing Migrations

There are few ways to run migrations.

1. Run node migration.js up. Runs all the pending up migrations.
2. Run node migration.js up 2. Runs 2 pending up migrations from the last position.
3. Run node migration.js down. Runs only 1 down migrations.
4. Run node migration.js refresh. Runs all down migrations followed by all up.

### Execute anonymous migrations

At times, few migrations need to run again or anonymously. There could be variety of reasons old migrations need to be executed or rollbacked. It can be done this way.

**Up migration**

node migration.js run 1500891087394\_create\_table\_users.js up

**Down migration**

node migration.js run 1500891087394\_create\_table\_users.js down

Since these are anonymous executions, no records are maintained for any executions.