

MongoDB con NodeJS

Introducción

Este documento servirá como representación de las características generales y consideraciones de la api de nóminas en desarrollo apartir de los conocimientos y aplicación práctica de bases de datos NoSQL, visualizadores de datos y procesamiento de datos.

En la actualidad la cantidad de información en el mundo crece con gran velocidad, es por esto que su almacenamiento se ha visto limitado en cuanto a características como el volumen, velocidad y escalabilidad de la información, siendo esto el motivo principal para problemas relevantes como la pérdida de información y sobrecostos.

Teniendo en cuenta lo anterior, se puede afirmar que la información tiene un enorme poder dentro de muchas áreas, específicamente en la toma de decisiones en cualquier organización, puesto que proporciona una ventaja competitiva o una oportunidad de negocio.

Una posible solución al problema de almacenamiento para grandes volúmenes de datos y su consulta de manera rápida y eficiente es la incorporación de nuevos enfoques tecnológicos en cuanto al almacenamiento, tal enfoque es conocido como bases de datos NoSQL.

MongoDB

El primer paso para implementar este documento es la construcción de un API REST que podemos consumir desde una aplicación web.

En éste tutorial explicaré como desarrollar servidor web que sirva una API RESTful usando para ello la tecnología de Node.js y MongoDB como base de datos.

Como framework para Node, se empleó **Express** en su versión 4.x y para conectarme a Mongo y mapear los modelos de datos, utilizaré Mongoose .

Iniciando el proyecto

El primer paso es crear un directorio en tu entorno local para la aplicación.

```
$ mkdir node-api-rest-example
```

```
$ cd node-api-rest-example
```

```
$ git init
```

El primer código que necesitamos escribir en una aplicación basada en Node es el archivo package.json. Éste archivo nos indica que dependencias vamos a utilizar en ella. Este archivo va en el directorio raíz de la aplicación:

Por lo tanto package.json tendrá:

```
{  
  "name": "node-api-rest-example",  
  "version": "2.0.0",  
  "dependencies": {  
    "mongoose": "~3.6.11",  
    "express": "^4.7.1",  
    "method-override": "^2.1.2",  
    "body-parser": "^1.5.1"  
  }  
}
```

Y ahora para descargar las dependencias escribimos lo siguiente en la consola y NPM (el gestor de paquetes de Node) se encargará de instalarlas.

```
$ npm install
```

Servidor Node.js

Creemos un fichero llamado app.js en el directorio raíz que será el que ejecute nuestra aplicación y arranque nuestro server. Crearemos en primer lugar un sencillo servidor web para comprobar que tenemos todo lo necesario instalado, y a continuación iremos escribiendo más código.

```
var express = require("express"),
    app = express(),
    bodyParser = require("body-parser"),
    methodOverride = require("method-override");
mongoose = require("mongoose");

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(methodOverride());

var router = express.Router();

router.get("/", function (req, res) {
  res.send("Hello World!");
});

app.use(router);

app.listen(3000, function () {
  console.log("Node server running on http://localhost:3000");
});
```

Las primeras líneas se encargan de incluir las dependencias que vamos a usar, algo así como los includes en C o PHP, o los import de Python. Importamos Express para facilitarnos crear el servidor y realizr llamadas HTTP. Con http creamos el servidor que posteriormente escuchará en el puerto 3000 de nuestro ordenador (O el que nosotros definamos).

Con `bodyParser` permitimos que pueda *parsear* JSON, `methodOverride()` nos permite implementar y personalizar métodos HTTP.

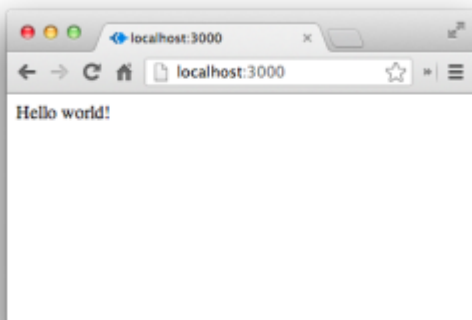
Podemos declarar las rutas con `app.route(nombre_de_la_ruta)` seguido de los verbos `.get()`, `.post()`, etc... y podemos crear una instancia para ellas con `express.Router()`. En este primer ejemplo vamos a hacer que sólo reciba una petición GET del navegador y muestre en el mismo la frase Hello World

Para ejecutar este pequeño código sólo tienes que escribir en consola lo siguiente y abrir un navegador con la url `http://localhost:3000`

```
$ node app.js
```

Node server running on `http://localhost:3000`

Si todo va bien, esto es lo que se verá:



Creando los modelos

En esta parte vamos a crear un modelo usando `Mongoose` para poder guardar la información en la base de datos siguiendo el esquema. Como base de datos vamos a utilizar `MongoDB`. `MongoDB` es una base de datos Open Source NoSQL orientada a documentos tipo JSON.

Para este ejemplo vamos a crear una base de datos de series de TV, por tanto vamos a crear un modelo (Archivo: `models/tvshow.js`) que incluya la información de una serie de TV, como pueden ser su título, el año de inicio, país de producción, una imagen promocional, número de temporadas, género y resumen del argumento:

```
var mongoose = require("mongoose"),
```

```
    Schema = mongoose.Schema;
```

```
var tvshowSchema = new Schema({
```

```
title: { type: String },
year: { type: Number },
country: { type: String },
poster: { type: String },
seasons: { type: Number },
genre: {
  type: String,
  enum: ["Drama", "Fantasy", "Sci-Fi", "Thriller", "Comedy"],
},
summary: { type: String },
});
```

```
module.exports = mongoose.model("TVShow", tvshowSchema);
```

Con esto ya podemos implementar la conexión a la base de datos en el archivo `app.js` añadiendo las siguientes líneas:

```
var mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/tvshows");
```

Quedando así el código de `app.js`:

```
var express = require("express"),
    app = express(),
    http = require("http"),
    server = http.createServer(app),
    mongoose = require("mongoose");

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(methodOverride());

var router = express.Router();
```

```
router.get("/", function (req, res) {  
  res.send("Hello World!");  
});  
  
app.use(router);  
  
mongoose.connect("mongodb://localhost/tvshows", function (err, res) {  
  if (err) {  
    console.log("ERROR: connecting to Database. " + err);  
  }  
  app.listen(3000, function () {  
    console.log("Node server running on http://localhost:3000");  
  });  
});
```

Para que esto funcione en nuestro entorno local, necesitamos tener instalado MongoDB.

Una vez hecho esto, para poder iniciar MongoDB debes ejecutar en otra terminal:

```
$ mongod
```

Con Mongo arrancado ya podemos ejecutar la aplicación como en la parte anterior con node app.js desde la terminal, si todo va bien tendremos algo en la pantalla como esto:

```
$ node app.js
```

```
Node server running on http://localhost:3000
```

```
Connected to Database
```

Ahora desde otra terminal, podemos entrar al shell de MongoDB y comprobar que la base de datos se ha creado correctamente. Para ello ejecutamos el comando mongo

```
$ mongo
```

```
MongoDB shell version: 2.4.1
```

connecting to: test

> use tvshows

switched to db tvshows

> show dbs

local 0.078125GB

tvshows (empty)

>_

Ya tenemos todo configurado y listo para albergar los datos, sólo nos queda crear las rutas que definirán las llamadas a la API para poder guardar y consultar la información

Implementando los controladores

los controladores de las rutas de nuestro API los vamos a crear en un archivo separado que llamaremos controllers/tvshows.js. Gracias a exports conseguimos modularizarlo y que pueda ser llamado desde el archivo principal de la aplicación. El código de a continuación es el comienzo del archivo con la primera función que será la que devuelva todos los registros almacenados:

//File: controllers/tvshows.js

```
var mongoose = require("mongoose");
```

```
var TVShow = mongoose.model("TVShow");
```

//GET - Return all tvshows in the DB

```
exports.findAllTVShows = function (req, res) {
```

```
  TVShow.find(function (err, tvshows) {
```

```
    if (err) res.send(500, err.message);
```

```
    console.log("GET /tvshows");
```

```
    res.status(200).jsonp(tvshows);
```

```
  });
```

```
};
```

De esta manera tan sencilla, al llamar a la función findAllTVShows se envía como respuesta toda la colección de tvshows almacenada y en formato JSON. Si

queremos que sólo nos devuelva un registro con un identificador único, tenemos que crear una función tal que la siguiente:

//GET - Return a TVShow with specified ID

```
exports.findByld = function(req, res) {  
  TVShow.findByld(req.params.id, function(err, tvshow) {  
    if(err) return res.send(500. err.message);  
  
    console.log('GET /tvshow/' + req.params.id);  
    res.status(200).jsonp(tvshow);  
  });  
};
```

Con las funciones find() y findByld() podemos buscar en la base de datos a partir de un modelo. Ahora desarrollaré el resto de funciones que permiten insertar, actualizar y borrar registros de la base de datos. La función de a continuación sería la correspondiente al método POST y lo que hace es añadir un nuevo objeto a la base de datos:

//POST - Insert a new TVShow in the DB

```
exports.addTVShow = function (req, res) {  
  console.log("POST");  
  console.log(req.body);  
  
  var tvshow = new TVShow({  
    title: req.body.title,  
    year: req.body.year,  
    country: req.body.country,  
    poster: req.body.poster,  
    seasons: req.body.seasons,  
    genre: req.body.genre,  
    summary: req.body.summary,  
  });
```



```

    tvshow.save(function (err, tvshow) {
      if (err) return res.status(500).send(err.message);
      res.status(200).jsonp(tvshow);
    });
  });
};

```

Primero creamos un nuevo objeto tvshow siguiendo el patrón del modelo, recogiendo los valores del cuerpo de la petición, lo salvamos en la base de datos con el comando .save() y por último lo enviamos en la respuesta de la función.

La siguiente función nos permitirá actualizar un registro a partir de un ID. Primero buscamos en la base de datos el registro dado el ID, y actualizamos sus campos con los valores que devuelve el cuerpo de la petición:

//PUT - Update a register already exists

```

exports.updateTVShow = function (req, res) {
  TVShow.findById(req.params.id, function (err, tvshow) {
    tvshow.title = req.body.petId;
    tvshow.year = req.body.year;
    tvshow.country = req.body.country;
    tvshow.poster = req.body.poster;
    tvshow.seasons = req.body.seasons;
    tvshow.genre = req.body.genre;
    tvshow.summary = req.body.summary;

    tvshow.save(function (err) {
      if (err) return res.status(500).send(err.message);
      res.status(200).jsonp(tvshow);
    });
  });
};
};

```

Y por último para completar la funcionalidad CRUD de nuestra API, necesitamos la función que nos permita eliminar registros de la base de datos y eso lo podemos hacer con el código de a continuación:

//DELETE - Delete a TVShow with specified ID

```
exports.deleteTVShow = function (req, res) {  
  TVShow.findById(req.params.id, function (err, tvshow) {  
    tvshow.remove(function (err) {  
      if (err) return res.status(500).send(err.message);  
      res.status(200).send();  
    });  
  });  
};
```

Como puedes ver, usamos de nuevo el método .findById() para buscar en la base de datos y para borrarlo usamos .remove() de la misma forma que usamos el .save() para salvar.

Ahora tenemos que unir estas funciones a las peticiones que serán nuestras llamadas al API. Volvemos a nuestro archivo principal, app.js y declaramos las rutas, siguiendo las pautas de Express v.4

```
var TVShowCtrl = require("./controllers/tvshows");
```

// API routes

```
var tvshows = express.Router();
```

```
tvshows
```

```
  .route("/tvshows")  
    .get(TVShowCtrl.findAllTVShows)  
    .post(TVShowCtrl.addTVShow);
```

```
tvshows
```

```
  .route("/tvshows/:id")
```

```
.get(TVShowCtrl.findById)
.put(TVShowCtrl.updateTVShow)
.delete(TVShowCtrl.deleteTVShow);
```

```
app.use("/api", tvshows);
```

El archivo completo **controllers/tvshows.js** lo puedes ver y descargar desde el repositorio en GitHub .

Probando el API REST

Antes de probarlo, debemos tener mongo y el servidor node de nuestra app corriendo. Una vez hecho esto introducimos los siguientes datos para hacer una llamada POST que almacene un registro en la base de datos.

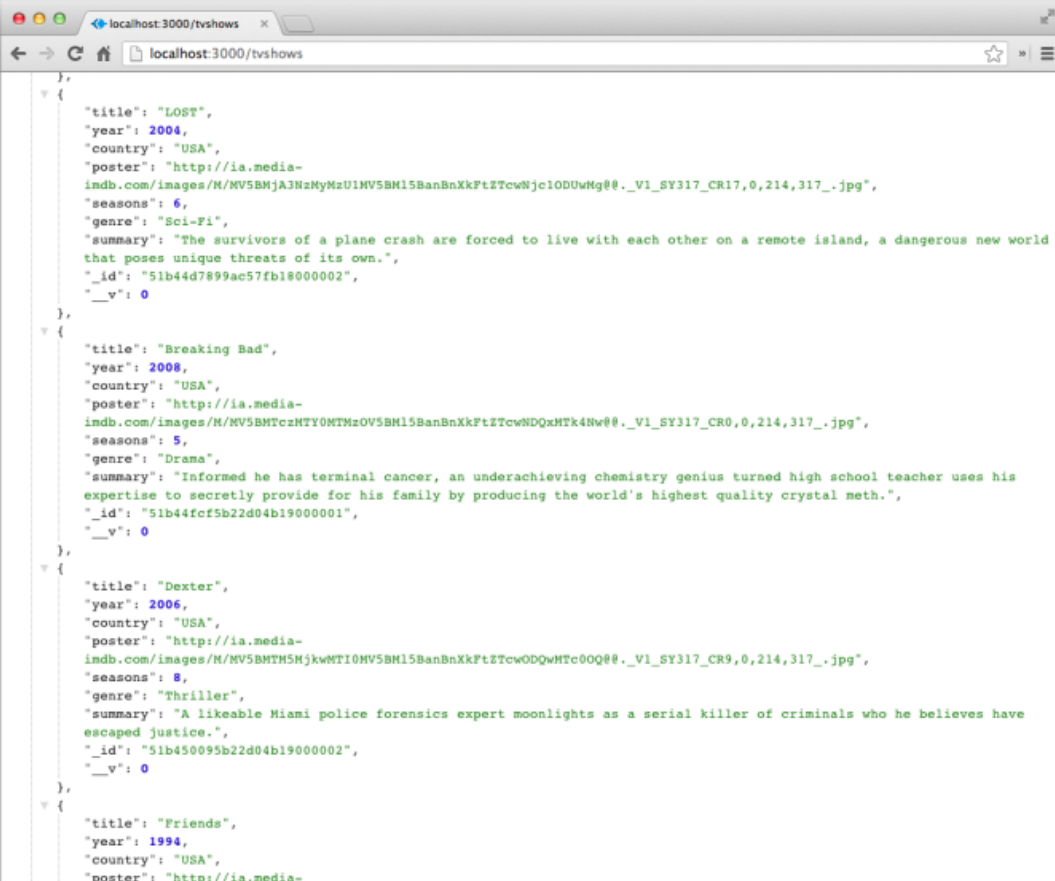
- **Target:** <http://localhost:3000/tvshow> (Dónde está ejecutándose la aplicación y la llamada al método POST que hemos programado)
- **Content-Type:** application/json (en los dos input que nos dan)
- **Request-Payload:** Aquí va el cuerpo de nuestra petición, con el objeto JSON siguiente (por ejemplo):

```
{
  "title": "LOST",
  "year": 2004,
  "country": "USA",
  "poster": "http://ia.media-
imdb.com/images/M/MV5BMjA3NzMyMzU1MV5BMT5BanBnXkFtZTcwNjc1ODUw
Mg@@._V1_SY317_CR17,0,214,317_.jpg",
  "seasons": 6,
  "genre": "Sci-Fi",
  "summary": "The survivors of a plane crash are forced to live with each other on a
remote island, a dangerous new world that poses unique threats of its own."
}
```



```
{ "title" : "LOST", "year" : 2004, "country" : "USA", "poster" : "http://ia.media-
imdb.com/images/M/MV5BMjA3NzMyMzU1MV5BMl5BanBnXkFtZTcwNjc1ODUw
Mg@@._V1_SY317_CR17,0,214,317_.jpg", "seasons" : 6, "genre" : "Sci-Fi",
"summary" : "The survivors of a plane crash are forced to live with each other on a
remote island, a dangerous new world that poses unique threats of its own.", "_id" :
ObjectId("51b44d7899ac57fb18000002"), "__v" : 0 }
```

Y ahí la tenemos, puedes probar a introducir alguna más, para tener mayor contenido con el que probar. Una vez introduzcas varios registros, puedes probar a llamarlos a través de la petición GET que hemos preparado: <http://localhost:3000/tvshows> la cuál te mostrará algo parecido a esto:



```
{
  "title": "LOST",
  "year": 2004,
  "country": "USA",
  "poster": "http://ia.media-
imdb.com/images/M/MV5BMjA3NzMyMzU1MV5BMl5BanBnXkFtZTcwNjc1ODUwMg@@._V1_SY317_CR17,0,214,317_.jpg",
  "seasons": 6,
  "genre": "Sci-Fi",
  "summary": "The survivors of a plane crash are forced to live with each other on a remote island, a dangerous new world
that poses unique threats of its own.",
  "_id": "51b44d7899ac57fb18000002",
  "__v": 0
},
{
  "title": "Breaking Bad",
  "year": 2008,
  "country": "USA",
  "poster": "http://ia.media-
imdb.com/images/M/MV5BMTczMTY0MTMzOV5BMl5BanBnXkFtZTcwNDQxMTk4Nw@@._V1_SY317_CR0,0,214,317_.jpg",
  "seasons": 5,
  "genre": "Drama",
  "summary": "Informed he has terminal cancer, an underachieving chemistry genius turned high school teacher uses his
expertise to secretly provide for his family by producing the world's highest quality crystal meth.",
  "_id": "51b44fcf5b22d04b19000001",
  "__v": 0
},
{
  "title": "Dexter",
  "year": 2006,
  "country": "USA",
  "poster": "http://ia.media-
imdb.com/images/M/MV5BMTM5MjkwMTI0MV5BMl5BanBnXkFtZTcwODQwMTc0OQ@@._V1_SY317_CR9,0,214,317_.jpg",
  "seasons": 8,
  "genre": "Thriller",
  "summary": "A likeable Miami police forensics expert moonlights as a serial killer of criminals who he believes have
escaped justice.",
  "_id": "51b450095b22d04b19000002",
  "__v": 0
},
{
  "title": "Friends",
  "year": 1994,
  "country": "USA",
  "poster": "http://ia.media-
```

También podemos llamar a un sólo registro gracias a la petición GET `tvshows/:id` que programamos, si ejecutamos por ejemplo **<http://localhost:3000/tvshow/51b44d7899ac57fb18000002>** nos devolverá un único objeto:



Los métodos restantes PUT y DELETE funcionan de manera parecida al POST sólo que hay que pasarte el valor del ID del objeto que queremos actualizar o borrar.

Conclusión

Como cualquier desarrollo, este tipo de arquitectura y diseño tiene sus beneficios e inconvenientes, pero podría destacar:

1. Es necesario realizar mantenimiento de los servidores donde se tienen instalados programas y aplicaciones. Es necesario instalar software, gestionar puertos de acceso o estar pendiente de las actualizaciones.
2. La sintaxis de consulta del lenguaje es compleja para un usuario no técnico.
3. Las funciones de la base de datos así como la documentación no son muy amplias.