# Mongoose / REST Api

Fuente: https://www.youtube.com/watch?v=gfP3aqV38q4&t=1410s

## 1 – Qué es Mongoose

Las aplicaciones que requieren conexión a bases de datos en general utilizan frameworks o librerías llamadas ORM (object-relational mapping). Estas permiten establecer conexiones y acceder a las bases de datos de manera más segura.

En Nodejs existe un módulo llamado Mongoose para trabajar con bases de datos MongoDB. Este no es un ORM, sino que se llama ODM ya que MongoBD es una base de datos no relacional y orientada a documentos (por eso es un object-document-mapping).

Una de las características fundamentales de Mongoose es que obliga a establecer modelos para los documentos que se van a almacenar en la base de datos, ya que al ser MongoDB una BD sin esquema, permite almacenar todo tipo de documentos en una misma colección, este en aplicaciones de gran tamaño si no se controla puede llegar a ser muy problemático.

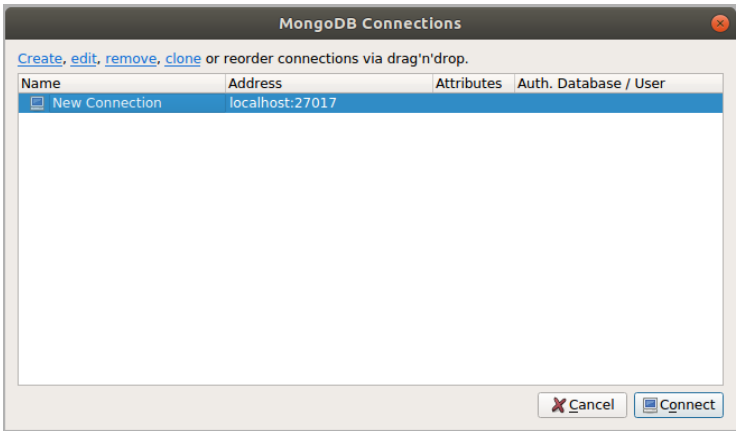
Además de facilitar y ordenar la conexión a la base de datos, mongoose también dispone de funciones para realizar consultas a la base de datos.

## 2 – Ejecutando mongoose

Para ejecutar mongoose es necesario primero tener MongoDB instalado y un servidor ejecutando. Esto se puede comprobar abriendo una terminal y ejecutando el comando **mongo** (en linux, para windows hay que iniciar el servidor y ejecutar el shell de mongo en dos consolas distintas), este muestra la direccion ip (en este caso es el localhost) y el puerto donde se encuentra la base de datos.

```
mongo
MongoDB shell version v3.6.3
connecting to: mongodb://127.0.0.1:27017
```

Tambien puede usarse la aplicación **Robo3t** que provee una interfaz gráfica para trabajar mas facilmente con mongoDB.



Una vez hecho esto creamos en una carpeta un archivo index.js y dentro de esa misma carpeta abrimos una terminal para instalar mongoose en el proyecto con el comando

**npm install mongoose**

Una vez instalado el modulo de mongoose lo requerimos desde la aplicación y nos conectamos a la base de datos indicando el protocolo (mongodb), la ip y el puerto, y el nombre de la base de datos. Si est última no existe se crea al insertar algún dato.

```
const mongoose = require('mongoose');

mongoose.connect('mongodb:/127.0.0.1:27017/appdb');
```

En caso de recibir algún error como el que sigue es necesario agregar el objeto indicado a las configuraciones de la conexión.

```
node index.js
(node:18856) DeprecationWarning: current URL string parser is deprecated, and will be remo
ved in a future version. To use the new parser, pass option { useNewUrlParser: true } to M
ongoClient.connect.
```

```
mongoose.connect('mongodb://localhost:27017/appdb',{
    useUnifiedTopology: true,
    useNewUrlParser: true
}
);
```

Ahora al ejecutar el archivo index.js no hace nada, vamos a agregar algunos eventos para mostrar por consola.

## 2.1 – Eventos

Mongoose provee una cantidad de eventos que se pueden escuchar para saber qué está pasando con la base de datos ( https://mongoosejs.com/docs/connections.html#connection-events ) por ejemplo si uno quiere mostrar por consola cada vez que se abre o cierra la conexión, cuando un usuario se conecta o desconecta, cuando ocurre algún error, etc. Los eventos se usan con el método **mongoose.connection.on(<evento>, callback)**

Por ejemplo establecemos una función que muestre por consola cuando se abre la conexión.

```
mongoose.connection.on('open', ()=> console.log(`BD conectada en ${uri}`));
```

```
[nodemon] starting `node index.js`
BD conectada en mongodb://localhost:27017/appdb
```

Para el manejo de errores podemos usar este mismo método (un error puede darse al pasar un dato con formato erroneo o al querer cargar en la bd un documento de mas de 16MB).

```
mongoose.connection.on('open', (err)=> console.log(err));
```

O también puede ser manejado con un **catch** al crear la conexión.

```
mongoose.connect( uri,{
    useUnifiedTopology: true,
    useNewUrlParser: true
}
).catch(err => console.log(err));
```

Otra forma de escuchar eventos es con el método **once**, que a diferencia de on solo se ejecuta una vez al ocurrir el evento indicado.

## 2.2 – Models

Como ya vimos, para trabajar con mongoose es necesario establecer el formato de los documentos que se van a guardar en cada colección de la bd. Por convención estas especie de clases, llamadas esquemas, se crean en un archivo propio para cada colección. La forma de declararlas es la siguiente

Supongamos que tenemos un negocio y necesitamos gestionar clientes y productos, entonces creamos los siguientes archivos.

1) Al archivo de configuración de la conexión lo llamamos **connection.js**

connection.js > ...

```
1  const mongoose = require('mongoose');
2
3  const uri = 'mongodb://localhost:27017/negocio'
4
5  const db = mongoose.connection;
6
7  mongoose.connect( uri,{
8      useUnifiedTopology: true,
9      useNewUrlParser: true
10 }
11 ).catch(err => console.log(err));
12
13 db.on('open', ()=> console.log(`BD conectada en ${uri}`));
```

2) creamos en la carpeta **models** el archivo **Cliente.js**

models > Cliente.js > ...

```
1  const { Schema, model} = require('mongoose');
2
3  const ClienteSchema = new Schema({
4      nombre: String,
5      activo: Boolean,
6      CUIT: String,
7      fechaAlta: Date
8  });
9
10 module.exports = model('Cliente', ClienteSchema);
```

En este archivo se requieren los objetos Schema y model de modulo mongoose. Luego se crea el esquema con los atributos que tendrá nuestro modelo de cliente, y por último se exporta el modelo.

3) Se crea en la raiz del proyecto un nuevo archivo **index.js**

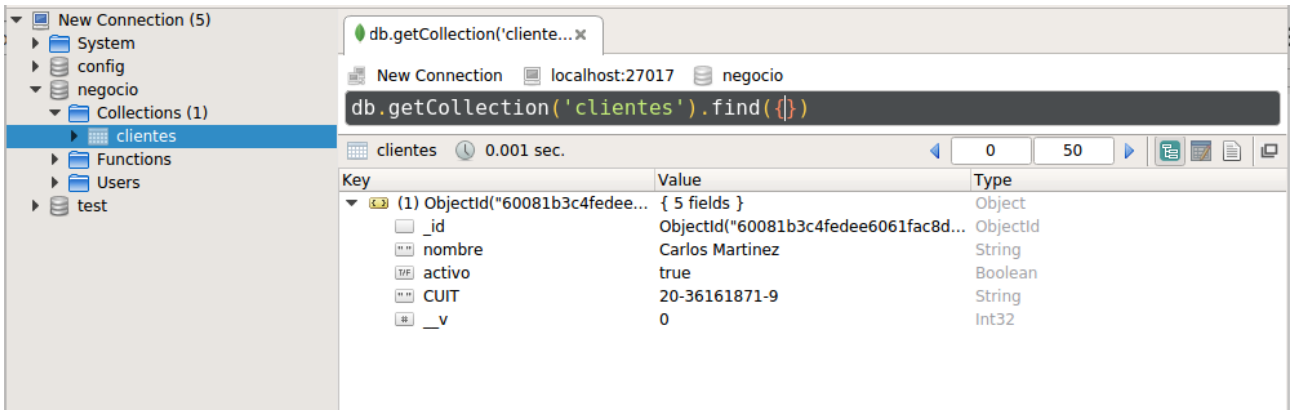
```
JS index.js > ...
1   require('./connection');
2
3   const Cliente = require('./models/Cliente');
4
5   const cliente1 = new Cliente({
6       nombre: 'Carlos Martinez',
7       activo: true,
8       CUIT: '20-36161871-9',
9       fechaAlta: new Date()
10  });
11
12  cliente1.save((err, doc)=>{
13      if (err) console.log(err);
14      console.log(doc);
15  })
```

Donde se importan los módulos de conexión y del modelo de cliente, se instancia un nuevo cliente, y se lo guarda en la base de datos con el método **save**, que tiene una función de callback que recibe como parámetros un error y el documento creado.

Al ejecutar el archivo index.js vemos como la conexión se estableció correctamente y se guardó el documento en la nueva base de datos

```
[nodemon] starting `node index.js`
BD conectada en mongodb://localhost:27017/negocio
{ _id: 60081b50d636076073bc00b3,
  nombre: 'Carlos Martinez',
  activo: true,
  CUIT: '20-36161871-9',
  fechaAlta: 2021-01-20T12:00:16.139Z,
  __v: 0 }
```

También podemos comprobarlo desde Robo3T



# 3 – ABM

Definimos un nuevo esquema para practicar las operaciones básicas sobre una base de datos, llamadas ABM( alta, baja, modificación) o CRUD (create, read, upload, delete).

Los atributos de los esquemas no solo pueden declararse con el tipo de dato, también pueden recibir un objeto json con distintos tipos de atributos, los mas usuarles son **type** (el tipo de dato), **default**(valor por defecto para que, en caso de ni indicarlo el documento genere igual el campo), valores máximos y mínimos, **required** (da error si no se asigna un valor), **unique** (no puede haber dos elementos con el mismo valor en ese campo)etc. Por ejemplo creamos el esquema de producto e indicamos que si no se indica un precio, este por defecto es 0. También se puede declarar funciones que validen el valor ingresado en uno de los campos.

models > JS Producto.js > ...

```
1  const {Schema, model} = require('mongoose');
2
3  const ProductoSchema = new Schema({
4      clave: {
5          type: String,
6          required: true,
7          unique: true
8      },
9      descripcion: String,
10     precio: {
11         type: Number,
12         default: 0,
13         validate(value){
14             if (value < 0) {
15                 throw new Error('El precio debe ser mayor o igual a cero');
16             }
17         }
18     }
19 })
```

Creamos en el directorio del proyecto las carpetas con los archivos **ABM/productoABM.js** y **test/testProductoABM.js**. En el archivo productoABM declaramos constantes para las funciones asíncronas que representan cada una de las operaciones ABM y las exportamos.

```
21 const traerProducto = async (clave)=>{};
22 const traerProductos = async ()=>{};
23 const bajaProducto = async (clave)=>{};
24 const modificarProducto = async(_id)=>{};
25
26 module.exports = {
27     traerProducto, //traer uno
28     traerProductos, //Traer todos
29     altaProducto,
30     bajaProducto,
31     modificarProducto
32 }
```

### Traer

En este caso usamos dos funciones, una para traer por clave y otra para traer todos los elementos de la colección.

```
const traerProducto = async (clave)=>{
    let retorno = await Producto.findOne({clave: clave});
    return retorno;
};

const traerProductos = async ()=>{
    let retorno = await Producto.find({});
    return retorno;
};
```

*IMPORTANTE: Cuando se crea la colección en mongoDB mongoose le pone el nombre en plural, asi que la colección se va a llamar **productos.** Sin embargo, al escribir las consultas el nombre que se usa tiene que ser el mismo del esquema, es decir, producto.

En este caso usamos funciones autoejecutables

* Ver por qué no funciona ejecutar las dos juntas

```
(async function(){
    let prod =  await prodABM.traerProducto('AG1');
    console.log(prod);
})()


(async function(){
    let prod =  await prodABM.traerProductos();
    console.log(prod);
})()
```

### Alta

En este archivo importamos el archivo de conexión y el modelo de producto y creamos una función asíncrona que recibe como parámetros los datos del producto, crea una instancia y la guarda en la base de datos.

```
const altaProducto = async (clave, descripcion, precio)=>{
    const prod = new Producto({
        clave,
        descripcion,
        precio
    });
    let retorno = false;
    try{
        await prod.save();
        retorno = true;
    } catch (err){
        throw new Error (err);
    }
    return retorno;
}
```

En el test usamos una promesa

```
test > JS testProductoABM.js > ...
1   const prodABM = require('../ABM/ProductoABM');
2
3   prodABM.altaProducto('AG1', 'Alfajor guaymallen', 30)
4       .then( ret => ret? console.log('producto guardado'): console.log('error'))
5       .catch( err => console.log(err));
```

### Baja

Para eliminar objetos suelen usarse principalmente cuatro médotos

- **deleteMany({clave: valor})**. Elimina todos los documentos con el valor indicado
- **findOneAndDelete({clave: valor})**. Elimina el primero que encuentre y retorna el elemento eliminado.
- **deleteOne({clave: valor})**. Busca todos y elimina solo uno.
- **FindById AndDelete(id)**. Elimina el dato con el id indicado y retorna el documento.

### Modificación

El update puede hacerse usando dos métodos distintos, primero con el método **findByIdAndUpdate()**, que recibe un criterio de búsqueda y la propiedad a modificar. En este caso se setean todas las propiedades ya que desde el test vamos a traer el objeto, hacer las modificaciones correspondientes y luego llamar a la modificación.

* Para este caso hay otros métodos similares como update() o findOneAndUpdate()

```
35  const modificarProducto = async(id, prod)=>{
36      let p = await Producto.findByIdAndUpdate(id,
37      {
38          clave: prod.clave,
39          descripcion: prod.descripcion,
40          precio: prod.precio
41      });
42      //return p; --> Retorna el elemento antes de ser modificado
43      return await Producto.findById(id);
44  };
```

La otra forma es traer el elemento, modificarlo, y usar el método **save()**

```
const modificarProducto = async(id, prod)=>{
    let p = await Producto.findById(id);
    p = prod;  //tambien puede hacerse para cada elemento del documento
    await p.save();
    return p;
};
```