Node.js

en el desarrollo de aplicaciones web y servicios

3. Acceso a bases de datos MongoDB con Node (I)
Conceptos básicos

Ignacio Iborra Baeza

Índice de contenidos

1. Introducción

En esta sesión daremos unas nociones básicas de cómo conectar y gestionar una base de datos MongoDB desde Node. Para los no iniciados en el tema, MongoDB es el principal representante, actualmente, de los sistemas de bases de datos No-SQL. Estos sistemas se han vuelto muy populares en los últimos años, y permiten dotar de persistencia a los datos de nuestra aplicación de una forma diferente a los tradicionales sistemas SQL.

En lugar de almacenar la información en tablas con sus correspondientes campos y registros, lo que haremos será almacenar estructuras de datos en formato BSON (similar a JSON), lo que facilita la integración con ciertas aplicaciones, como las aplicaciones Node.

1.1. Algunos conceptos de bases de datos No-SQL

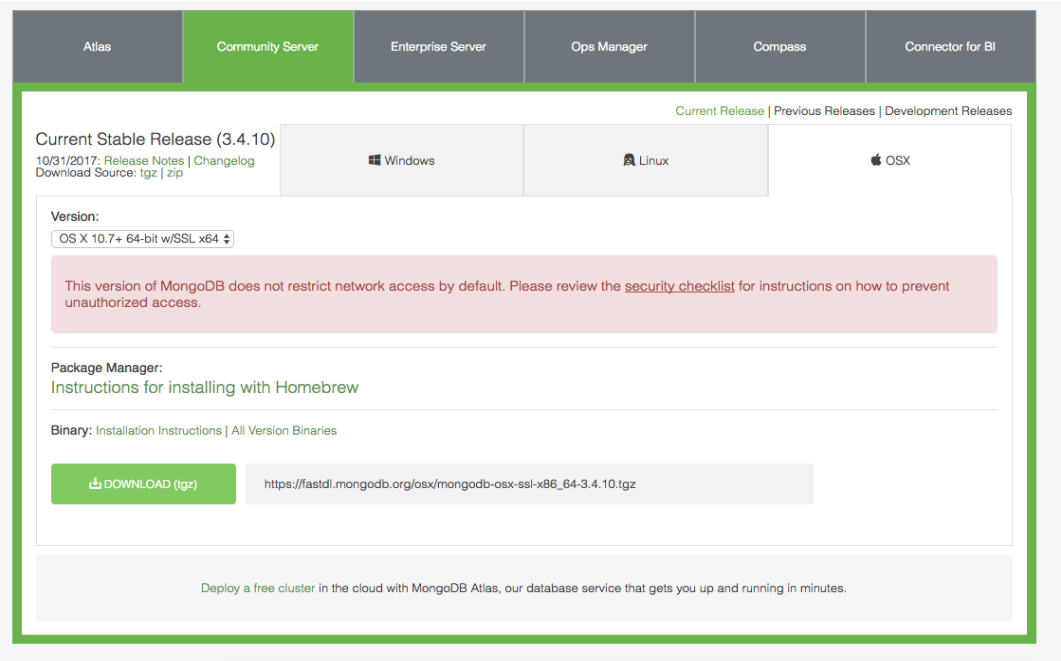
Las bases de datos No-SQL tienen algunas similitudes y diferencias con las tradicionales bases de datos SQL. Entre las similitudes, las dos trabajan con bases de datos, es decir, lo que creamos en uno u otro gestor es siempre una base de datos, pero la principal diferencia radica en cómo se almacenan los datos. En una base de datos SQL, la información se almacena en forma de tablas, mientras que en una No-SQL lo que se almacena se denominan **colecciones** (arrays de objetos en formato BSON, en el caso de Mongo). Las tablas están compuestas de registros (cada fila de la tabla), mientras que las colecciones se componen de **documentos** (cada objeto de la colección). Finalmente, cada registro de una tabla SQL tiene una serie de campos fijos (todos los registros de la tabla tienen los mismos campos), mientras que en una colección No-SQL, cada documento puede tener un conjunto diferente de **propiedades** (que también se suelen llamar *campos*). En cualquier caso, lo habitual es que los documentos de una misma colección compartan las mismas propiedades.

1.2. Instalando y configurando MongoDB

Como decíamos, el ejemplo más representativo de los sistemas No-SQL actualmente vigentes es MongoDB, un sistema de bases de datos de código abierto y multiplataforma, que podemos instalar en sistemas Windows, Mac OSX o Linux.

Veremos cómo instalar MongoDB manualmente en estos tres sistemas, y también veremos cómo podemos configurarlo como un servicio en nuestra máquina virtual. En el caso de querer hacer esto último, podemos saltarnos estos pasos que vienen a continuación, e ir directamente al apartado en cuestión.

Para proceder a descargar e instalar MongoDB, accedemos a su web oficial, y en concreto a la sección de descargas. Nos interesa descargar la versión *Community*, que es gratuita y no requiere registro.

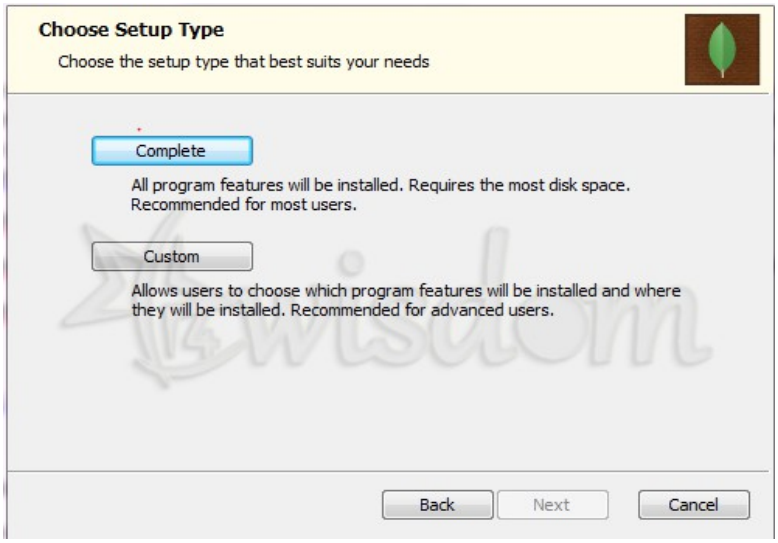


1.2.1. Instalación para Linux y Mac OS X

Si estamos utilizando un sistema Linux o Mac OS X, simplemente especificamos el sistema operativo que tenemos en la lista desplegable del formulario de descarga, y hacemos clic en el enlace para descargar. Lo que obtendremos será un archivo *.tar.gz* que podemos descomprimir. Dentro contiene los ejecutables para poner en marcha el servidor y conectar con él. Conviene mover (y renombrar, si se quiere) la carpeta que se genera al extraer a un lugar apropiado (a la carpeta personal, por ejemplo). En estos apuntes asumiremos que se ha instalado en la carpeta "mongo", dentro de nuestra carpeta personal, pero puede ser cualquier otra.

1.2.2. Instalación para Windows

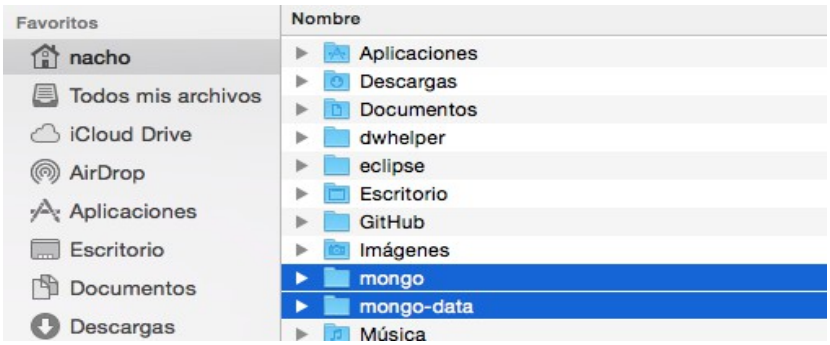
En el caso de usuarios Windows, también debemos elegir la versión que prefiramos del desplegable, aunque en este caso todas se refieren a Windows Server, no hay mucho donde elegir, y la opción por defecto suele ser la adecuada. Lo que se descarga es un instalador (archivo *.msi*) que lanza un asistente para instalar MongoDB en el sistema. Podemos seguir los pasos y elegir la instalación completa.



Conviene recordar la carpeta donde se instala (normalmente en *Archivos de programa\MongoDB\Server\X.Y*, siendo *X.Y* el número de versión que hayamos descargado). Deberemos acceder manualmente después a esta carpeta para iniciar el servidor.

1.2.3. Establecer la carpeta para las bases de datos

En cualquier caso, y antes de iniciar el servidor y crear o manipular bases de datos, debemos definir en qué carpeta se van a almacenar. Lo habitual es crear una carpeta llamada *mongo-data* en nuestra carpeta de usuario (aunque el nombre y la ubicación pueden ser los que queramos). En el caso de Mac OS X, por ejemplo, podríamos tener tanto la carpeta con *MongoDB* como la carpeta para los datos en la misma carpeta personal:



1.2.4. Poner en marcha el servidor

Para iniciar el servidor MongoDB en cualquier sistema, debemos acceder desde un terminal a la subcarpeta *bin* de la carpeta de instalación de MongoDB, y ejecutar desde el propio terminal el comando `mongod`. indicando en el parámetro `--dbpath` la ruta hacia la carpeta que hemos creado para almacenar los datos. Por ejemplo, desde Mac OS X o Linux podríamos poner algo como esto:

```
./mongod --dbpath /Users/nacho/mongo-data
```

En el caso de Windows, deberemos ejecutar el comando `mongod.exe` desde el terminal con los mismos parámetros. Se mostrarán unos cuantos mensajes por la consola, y el último indicará que MongoDB queda a la espera de conexiones en el puerto 27017:

```
2018-04-15T01:12:27.195+0100 I NETWORK  [thread1] waiting for
connections on port 27017
```

1.2.5. Instalar MongoDB como servicio

Existe la posibilidad de instalar MongoDB como un servicio que se inicie de forma automática, tanto en Linux como en Mac o Windows, aunque para estos últimos existe alguna traba adicional. En el caso de Mac, tenemos que tener instalado el programa *Homebrew*, que no viene por defecto en el sistema, y en el caso de Windows hay que definir a mano algunos parámetros del servicio. Al no ser el objetivo principal de este curso, y dado que podemos ponerlo en marcha sin problemas con las indicaciones anteriores, no trataremos estas dos opciones aquí, pero sí veremos cómo instalarlo como servicio en nuestra máquina virtual Linux.

Para instalar MongoDB como servicio en Linux, y como alternativa a la instalación manual vista anteriormente, podemos ejecutar estos comandos, como usuario *root*:

```
apt-get update
```

```
apt-get install mongodb
```

Tras estos comandos, ya tendremos el servidor disponible, y podremos iniciarlo, detenerlo o reiniciarlo con estos comandos:

```
/etc/init.d/mongodb start
/etc/init.d/mongodb stop
/etc/init.d/mongodb restart
```

1.2.6. Solución de problemas con libssl

En algunas versiones de Linux, como Debian 9, puede que dé algún problema al iniciar el servidor Mongo, relacionado con una librería SSL, y necesitemos instalarla siguiendo estos pasos:

- Descargar la librería desde este enlace
- Ejecutar este comando en la carpeta donde hemos descargado la librería:

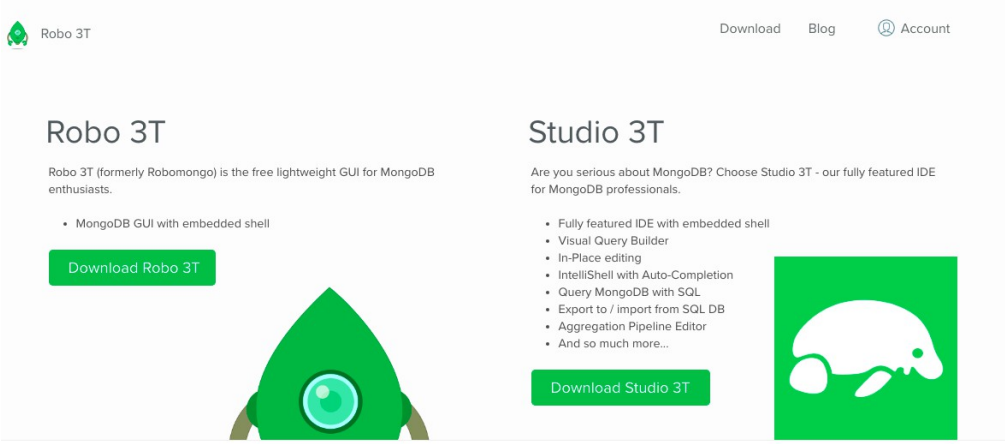
  ```
  dpkg -i libssl1.0.0_1.0.1t-1+deb8u7_amd64.deb
  ```

En cualquier caso, conviene primero intentar iniciar el servidor de alguna de las formas explicadas con anterioridad, y sólo en caso de que dé el mensaje de error relacionado con la librería, intentar este último paso.

1.3. Una GUI para MongoDB: RoboMongo/Robo 3T

Aunque podemos emplear el terminal para conectar con MongoDB y hacer la mayoría de las operaciones habituales de gestión de una base de datos, emplear dicho terminal puede resultar tedioso en ocasiones.

Para una gestión más amigable, podemos emplear un gestor gráfico como RoboMongo (actualmente llamado Robo 3T). Es un gestor gratuito y multiplataforma, que se puede descargar desde su web oficial.



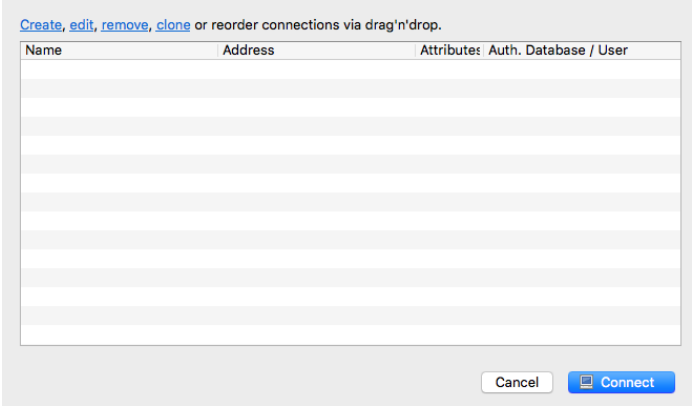
1.3.1. Descarga e instalación

En la sección de descargas (*Download*) podremos elegir la versión que se adapte a nuestro sistema operativo (Windows, Mac o Linux):

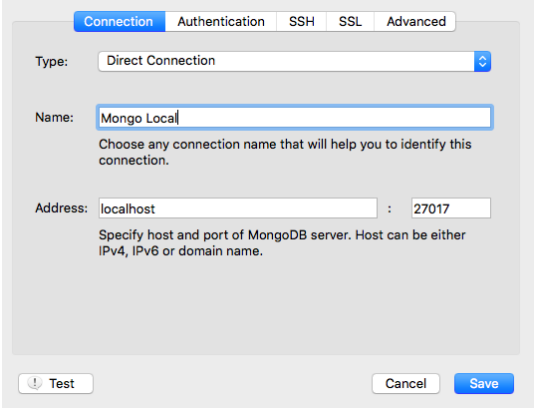
En el caso de Mac o Windows, disponemos de un asistente que instala Robo 3T. El de Mac es uno de esos asistentes donde debemos arrastrar el programa a la carpeta de Aplicaciones. Para Linux, tenemos un archivo *.tar.gz* que podemos descomprimir, y después podemos ejecutar el programa desde la subcarpeta *bin*.

1.3.2. Puesta en marcha y conexión con la base de datos

Una vez descargado e instalado Robo 3T, al ponerlo en marcha veremos un panel inicial de configuración de conexiones:



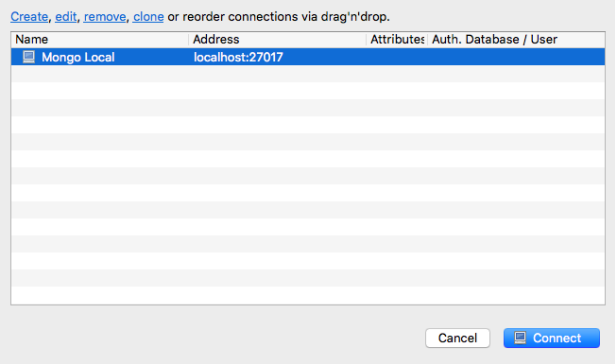
Aquí debemos configurar la conexión a nuestro servidor `mongod` (suponiendo que lo tengamos iniciado como se ha explicado anteriormente). Pulsamos en el enlace *Create* de la parte superior izquierda y configuramos los parámetros de conexión (también podemos editar conexiones o borrarlas desde estos enlaces):



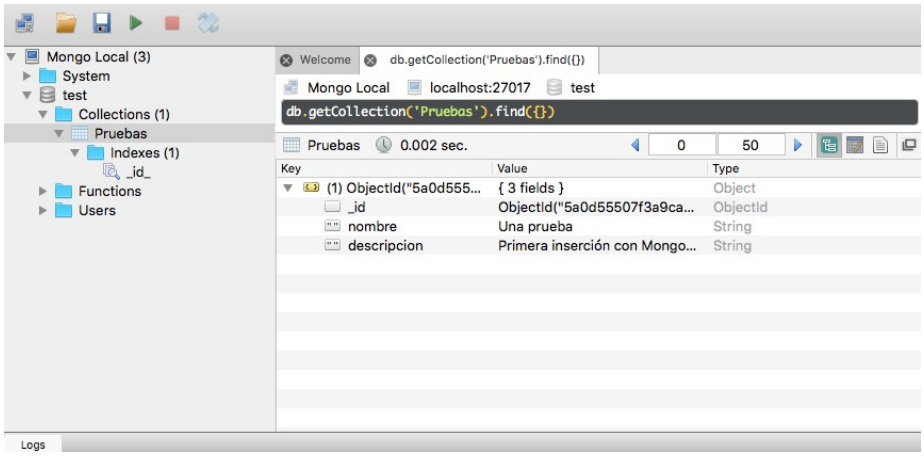
- Nombre de la conexión, que puede ser por ejemplo "Mongo Local"

- La dirección y puerto por defecto son los apropiados (*localhost* y 27017).

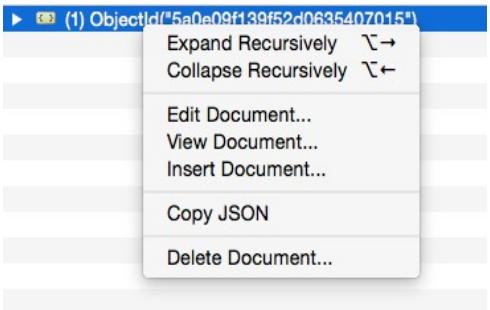
Guardamos los cambios, y desde el panel principal de conexiones seleccionamos la que hemos creado y pulsamos en el botón *Connect*.



Al conectar, en el panel izquierdo tendremos el explorador para examinar los elementos del servidor Mongo. Cuando creemos una base de datos, aparecerá en el listado, junto con las tablas o colecciones que contiene. Haciendo clic en una colección, podemos ver a la derecha los documentos que contiene.



Además, haciendo clic derecho en el panel de la derecha, podemos insertar nuevos documentos en la colección, o borrar/modificar los existentes (haciendo el clic derecho sobre ellos).



1.4. Importar/exportar colecciones de datos

Robo 3T no ofrece (por el momento) la opción de importar o exportar colecciones desde la GUI, pero podemos emplear el terminal para ello.

1.4.1. Exportación de colecciones

Para **exportar** una colección, accedemos a la subcarpeta *bin* de nuestra instalación de Mongo, y ejecutamos este comando:

```
./mongoexport --db nombre_bd --collection nombre_coleccion --out fichero
```

donde *nombre_bd* será el nombre de la base de datos a emplear, *nombre_coleccion* será el nombre de la colección a exportar y *fichero* será el fichero donde exportarla (con ruta relativa a la carpeta actual, o absoluta). Por ejemplo:

```
./mongoexport --db test --collection Pruebas --out Pruebas.json
```

1.4.2. Importación de colecciones

Para **importar** una colección previamente exportada, en primer lugar deberemos asegurarnos de que existe la base de datos donde la queramos importar (podemos crearla desde Robo 3T haciendo clic derecho en el servidor del panel izquierdo, y eligiendo *Create database*). Si dentro de la base de datos hay una colección con el mismo nombre que la que queremos importar, los datos se añadirán a la colección existente, salvo que la borremos antes de importar, o los *id* coincidan con algunos ya existentes. El comando para importar los datos es:

```
./mongoimport --db nombre_bd --collection nombre_coleccion --file fichero
```

donde *nombre_bd* es el nombre de la base de datos donde importar, *nombre_coleccion* es el nombre de la colección donde importar y *file* es el fichero de donde leer los datos. Por ejemplo:

```
./mongoimport --db otrotest --collection Pruebas --file Pruebas.json
```

2. La librería "mongoose"

Existen varias librerías en el repositorio oficial de NPM para gestionar bases de datos MongoDB, pero la más popular es Mongoose. Permite acceder de forma fácil a las bases de datos y, además, definir *esquemas,* una estructura de validación que determina el tipo de dato y rango de valores adecuado para cada campo de los documentos de una colección. Así, podemos establecer si un campo es obligatorio o no, si debe tener un valor mínimo o máximo, etc. En la web oficial de Mongoose podemos consultar algunos ejemplos de definición de esquemas y documentación adicional.

2.1. Primeros pasos

A lo largo de este apartado vamos a hacer algunas pruebas con Mongoose en un proyecto que llamaremos "PruebaMongo". Podemos crearlo ya en nuestra carpeta "ProyectosNode/Pruebas". Después, definiremos el archivo "package.json" con el comando `npm init`, y posteriormente instalaremos *mongoose* en el proyecto con el comando `npm install`.

```
npm init
```

```
npm install mongoose
```

Una vez instalado, necesitamos incorporarlo al código del proyecto con la correspondiente instrucción `require`. Crea un archivo fuente "index.js" en este proyecto de pruebas, e incorpora la libraría de este modo::

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

2.2. Conectar al servidor

Para conectar con el servidor Mongo (y suponiendo que lo tenemos iniciado, siguiendo los pasos indicados en el primer apartado de esta sesión), necesitamos llamar a un método llamado `connect`, dentro del objeto `mongoose` que hemos incorporado. Le pasaremos la URL de la base de datos como parámetro. Por ejemplo, con esta instrucción conectamos con una base de datos llamada "contactos" en el servidor local:

```
mongoose.connect('mongodb://localhost:27017/contactos');
```

No os preocupéis por que la base de datos no exista. Se creará automáticamente tan pronto como añadamos datos en ella.

> **AVISO**: en versiones anteriores de Mongoose, el método `connect` tenía una variante recomendada, que consistía en añadir un segundo parámetro `{useMongoClient: true}`. Este segundo parámetro ha dejado de ser necesario en las versiones recientes de la librería.

2.2.1. Establecer el motor de promesas

Como veremos a continuación, los métodos que ofrece Mongoose para obtener listados, insertar, borrar, etc, funcionan a base de promesas, que lanzan la tarea de forma asíncrona y después recogen el resultado. Por lo tanto, además de la instrucción de conexión vista antes, necesitamos establecer un parámetro de configuración que indique qué tipo de promesas va a emplear Mongoose.

Este paso puede parecer extraño, pero existen distintos tipos de promesas implementadas en las librerías de Javascript, y Mongoose nos permite utilizar cualquiera de ellas. Si simplemente queremos emplear la promesa por defecto de Javascript, basta con añadir esta línea de código que tenéis destacada en negrita (normalmente, justo antes o después de conectar con la base de datos):

```
mongoose.Promise = global.Promise;
mongoose.connect('mongodb://localhost:27017/contactos');
```

En este punto, puedes realizar el Ejercicio 1 de los propuestos al final de la sesión.

2.3. Modelos y esquemas

Como comentábamos antes, la librería Mongoose permite definir la estructura que van a tener los documentos de las distintas colecciones de la base de datos. Para ello, se definen **esquemas** (*schemas*) y se asocian a **modelos** (las colecciones correspondientes en la base de datos).

2.3.1. Definir los esquemas

Para definir un esquema, necesitamos crear una instancia de la clase `Schema` de Mongoose. Por lo tanto, crearemos este objeto, y en esa creación definiremos los atributos que va a tener la colección correspondiente, junto con el tipo de dato de cada atributo.

En el caso de la base de datos de contactos propuesta para estas pruebas, podemos definir un esquema para almacenar los datos de cada contacto: nombre, número de teléfono y edad, por ejemplo. Esto lo haríamos de esta forma:

```
let contactoSchema = new mongoose.Schema({
    nombre: String,
    telefono: String,
    edad: Number
});
```

Los tipos de datos disponibles para definir el esquema son:

- Textos (`String`)
- Números (`Number`)
- Fechas (`Date`)
- Booleanos (`Boolean`)
- Arrays (`Array`)
- Otros (veremos algunos más adelante): `Buffer, Mixed, ObjectId`

2.3.2. Aplicar el esquema a un modelo

Una vez definido el esquema, necesitamos aplicarlo a un modelo para asociarlo así a una colección en la base de datos. Para ello, disponemos del método `model` en Mongoose. Como primer parámetro, indicaremos el nombre de la colección a la que asociar el esquema. Como segundo parámetro, indicaremos el esquema a aplicar (objeto de tipo `Schema` creado anteriormente):

```
let Contacto = mongoose.model('contactos', contactoSchema);
```

NOTA: si indicamos un nombre de modelo en singular, Mongoose automáticamente creará la colección con el nombre en plural. Este plural no siempre será correcto, ya que lo que hace es simplemente añadir una "s" al final del nombre del modelo, si no se la hemos añadido nosotros.

2.3.3. Restricciones y validaciones

Si definimos un esquema sencillo como el ejemplo de contactos anterior, permitiremos que se añada cualquier tipo de valor a los campos de los documentos. Así, por ejemplo, podríamos tener contactos sin nombre, o con edades negativas. Pero con Mongoose podemos proporcionar mecanismos de validación que permitan descartar de forma automática los documentos que no cumplan las especificaciones.

En la documentación oficial de Mongoose podemos encontrar una descripción detallada de los diferentes validadores que podemos aplicar. Aquí nos limitaremos a describir los más importantes o habituales:

- El validador **`required`** permite definir que un determinado campo es obligatorio.
- El validador **`default`** permite especificar un valor por defecto para el campo, en el caso de que no se especifique ninguno.
- Los validadores **`min`** y **`max`** se utilizan para definir un rango de valores (mínimo y/o máximo) permitidos para datos de tipo numérico.
- Los validadores **`minlength`** y **`maxlength`** se emplean para definir un tamaño mínimo o máximo de caracteres, en el caso de cadenas de texto.
- El validador **`unique`** indica que el campo en cuestión no admite duplicados (sería una clave alternativa, en un sistema relacional).
- El validador **`match`** se emplea para especificar una expresión regular que debe cumplir el campo (aquí tenéis más información al respecto).
- ...

Volvamos a nuestro esquema de contactos. Vamos a establecer que el nombre y el teléfono sean obligatorios, y sólo permitiremos edades entre 18 y 120 años (inclusive). Además, el nombre tendrá una longitud mínima de 1 carácter, y el teléfono estará compuesto por 9 dígitos, empleando una expresión regular, y será una clave única. Podemos emplear algún validador más, como por ejemplo `trim`, para limpiar los espacios en blanco al inicio y final de los datos de texto. Con todas estas restricciones, el esquema queda de esta forma:

```
let contactoSchema = new mongoose.Schema({
    nombre: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    telefono: {
        type: String,
        required: true,
        unique: true,
```

```
        trim: true,
        match: /^\d{9}$/
    },
    edad: {
        type: Number,
        min: 18,
        max: 120
    }
});
```

Ya tenemos establecida la conexión a la base de datos, y el esquema de los datos que vamos a utilizar. Podemos empezar a realizar algunas operaciones básicas contra dicha base de datos.

En este punto, puedes realizar el Ejercicio 2 de los propuestos al final de la sesión.

2.4. Añadir documentos

Si queremos insertar o añadir un documento en una colección, debemos crear un objeto del correspondiente modelo, y llamar a su método `save`. Este método devuelve una promesa, por lo que emplearemos:

- Un bloque de código `then` para cuando la operación haya ido correctamente. En este bloque, recibiremos como resultado el objeto que se ha insertado, pudiendo examinar los datos del mismo si se quiere.
- Un bloque de código `catch` para cuando la operación no haya podido completarse. Recibiremos como parámetro un objeto con el error producido, que podremos examinar para obtener más información sobre el mismo.

Así añadiríamos un nuevo contacto a nuestra colección de pruebas:

```
let contacto1 = new Contacto({
    nombre: "Nacho",
    telefono: "966112233",
    edad: 39
});
contacto1.save().then(resultado => {
    console.log("Contacto añadido:", resultado);
}).catch(error => {
    console.log("ERROR añadiendo contacto:", error);
});
```

Añade este código al archivo "index.js" de nuestro proyecto "PruebaMongo", tras la conexión a la base de datos y la definición del esquema y modelo. Ejecuta la aplicación, y echa un vistazo al resultado que se devuelve cuando todo funciona correctamente::

```
{ __v: 0,
  nombre: 'Nacho',
  telefono: '966112233',
  edad: 39,
  _id: 5a12a2e0e6219d68c00c6a00 }
```

Observa que obtenemos los mismos campos que definimos en el esquema (nombre, teléfono y edad), y dos campos adicionales que no hemos especificado:

- `__v` hace referencia a la versión del documento. Inicialmente, todos los documentos parten de la versión 0 cuando se insertan, y luego esta versión puede modificarse cuando hagamos alguna actualización del documento, como veremos después.
- `_id` es un código autogenerado por Mongo, para cualquier documento de cualquier colección que se tenga. Analizaremos en qué consiste este *id* y su utilidad en breve.

Vayamos ahora a Robo 3T y examinemos las bases de datos en el panel izquierdo. Si clicamos en la colección de "contactos", veremos el nuevo contacto añadido en el panel derecho:

Puedes realizar ahora el Ejercicio 3 del final de la sesión.

Si intentamos insertar un contacto incorrecto, saltaremos al bloque `catch`. Por ejemplo, este contacto es demasiado viejo, según la definición del esquema:

```
let contacto2 = new Contacto({
    nombre: "Matuzalem",
    telefono: "965123456",
    edad: 200
});
contacto2.save().then(resultado => {
    console.log("Contacto añadido:", resultado);
}).catch(error => {
    console.log("ERROR añadiendo contacto:", error);
});
```

Si echamos un vistazo al error producido, veremos mucha información, pero entre toda esa información hay un atributo llamado `ValidationError` con la información del error:

```
ValidationError: contacto validation failed: edad: Path `edad` (200) is more than maximum allowed value (120)
```

Sobre este aspecto, tienes propuesto (de carácter optativo), el Ejercicio 4 del final de la sesión.

2.4.1. Sobre el *id* automático

Como has podido ver en las pruebas de inserción anteriores, cada vez que se añade un documento a una colección se le asigna automáticamente una propiedad llamada `_id` con un código autogenerado. A diferencia de otros sistemas de gestión de bases de datos

(como MariaDB/MySQL, por ejemplo), este código no es autonumérico, sino que es una cadena. De hecho, es un texto de 12 bytes que almacena información importante:

- El tiempo de creación de documento (*timestamp*), con lo que podemos obtener el momento exacto (fecha y hora) de dicha creación
- El ordenador que creó el documento. Esto es particularmente útil cuando queremos escalar la aplicación y tenemos distintos servidores Mongo accediendo a la misma base de datos. Podemos identificar cuál de todos los servidores fue el que creó el documento.
- El proceso concreto del sistema que creó el documento
- Un contador aleatorio, que se emplea para evitar cualquier tipo de duplicidad, en el caso de que los tres valores anteriores coincidan en el tiempo.

Existen métodos específicos para extraer parte de esta información, en concreto el momento de creación, pero no los utilizaremos por el momento.

A pesar de disponer de esta enorme ventaja con este *id* autogenerado, podemos optar por crear nuestros propios *ids* y no utilizar los de Mongo (aunque ésta no es una buena idea):

```
let contactoX = new Contacto({_id:2, nombre:"Juan",
telefono:"611885599"});
```

2.5. Buscar documentos

Si queremos buscar cualquier documento, o conjunto de documentos, en una colección, podemos emplear diversos métodos.

2.5.1. Búsqueda genérica con "find"

La forma más general de obtener documentos consiste en emplear el método **`find`** asociado al modelo en cuestión. Podemos emplearlo sin parámetros (con lo que obtendremos todos los documentos de la colección):

```
Contacto.find().then(resultado => {
    console.log(resultado);
}).catch (error => {
    console.log("ERROR:", error);
});
```

Observa que, en caso de resultado satisfactorio, obtendremos dicho resultado en la cláusula `then`, y en caso de error, lo obtendremos en la cláusula `catch`, como ocurrirá con todas las demás operaciones que hagamos.

2.5.2. Búsqueda parametrizada con "find"

Podemos también pasar como parámetro a `find` un conjunto de criterios de búsqueda. Por ejemplo, para buscar contactos cuyo nombre sea "Nacho" y la edad sea de 29 años, haríamos esto:

```
Contacto.find({nombre: 'Nacho', edad: 29}).then(resultado => {
    console.log(resultado);
}).catch (error => {
    console.log("ERROR:", error);
});
```

NOTA: cualquier llamada a `find` devolverá un array de resultados, aunque sólo se haya encontrado uno, o ninguno. Es importante tenerlo en cuenta para luego saber cómo acceder a un elemento concreto de dicho resultado. El hecho de no obtener resultados no va a provocar un error (no se saltará al `catch` en ese caso)

También podemos emplear algunos operadores de comparación en el caso de no buscar datos exactos. Por ejemplo, esta consulta obtiene todos los contactos cuyo nombre sea "Nacho" y las edades estén comprendidas entre 18 y 40 años:

```
Contacto.find({nombre:'Nacho', edad: {$gte: 18, $lte: 40}}).then(resultado => {
    console.log('Resultado de la búsqueda:', resultado);
}).catch(error => {
    console.log('ERROR:', error);
});
```

Aquí podéis encontrar un listado detallado de los operadores que podéis utilizar en las búsquedas.

2.5.3. Otras opciones: "findOne" o "findById"

Existen otras alternativas que podemos utilizar para buscar documentos concretos (y no un conjunto o lista de ellos). Se trata de los métodos **`findOne`** y **`findById`**. El primero se emplea de forma similar a `find`, con los mismos parámetros de filtrado, pero sólo devuelve un documento que concuerde con esos criterios (no un array). Por ejemplo:

```
Contacto.findOne({nombre:'Nacho', edad: 39}).then(resultado => {
    console.log('Resultado de la búsqueda:', resultado);
}).catch(error => {
    console.log('ERROR:', error);
});
```

El método `findById` se emplea, como su nombre indica, para buscar un documento dado su id (el *id* autogenerado por Mongo). Por ejemplo:

```
Contacto.findById('5ab2dfb06cf5de1d626d5c09').then(resultado => {
    console.log('Resultado de la búsqueda por ID:', resultado);
}).catch(error => {
    console.log('ERROR:', error);
});
```

En estos casos, si la consulta no produce ningún resultado, obtendremos `null` como respuesta, pero tampoco se activará la cláusula `catch` por ello.

Con lo visto hasta ahora, intenta realizar el Ejercicio 5 de los del final de la sesión

2.6. Borrar documentos

Para eliminar documentos de una colección, podemos emplear los métodos estáticos:

- **`remove`**, que elimina los documentos que cumplan los criterios indicados como parámetro. Estos criterios se especifican de la misma forma que hemos visto para el método `find`. Si no se especifican parámetros, se eliminan TODOS los documentos de la colección.

  ```
  Contacto.remove({nombre: 'Nacho'}).then(resultado => {
      console.log(resultado);
  }).catch (error => {
      console.log("ERROR:", error);
  ```

```
});
```

El resultado que se obtiene en este caso contiene múltiples propiedades. En la propiedad `result` podemos consultar el número de filas afectadas (n), y el resultado de la operación (ok).

```
CommandResult {
  result: { n: 1, ok: 1 },
  connection:
  ...
```

- **`findOneAndRemove`**, que busca el documento que cumpla el patrón (o el primero que lo cumpla) y lo elimina. Además, obtiene el documento eliminado en el resultado, con lo que podríamos deshacer la operación a posteriori, si quisiéramos, volviéndolo a añadir.

```
Contacto.findOneAndRemove({nombre: 'Nacho'})
.then(resultado => {
    console.log("Contacto eliminado:", resultado);
}).catch (error => {
    console.log("ERROR:", error);
});
```

Observad que, en este caso, el parámetro `resultado` es directamente el objeto eliminado.

- **`findByIdAndRemove`**, que busca el documento con el *id* indicado y lo elimina. También obtiene como resultado el objeto eliminado.

```
Contacto.findByIdAndRemove('5a16fed09ed79f03e490a648')
.then(resultado => {
    console.log("Contacto eliminado:", resultado);
}).catch (error => {
    console.log("ERROR:", error);
});
```

En el caso de estos dos últimos métodos, si no se ha encontrado ningún elemento que cumpla el criterio de filtrado, se devolverá *null* como resultado (es decir, no se activará la cláusula `catch` por este motivo).

Prueba ahora con el Ejercicio 6 de los propuestos al final de la sesión.

2.7. Modificaciones o actualizaciones de documentos

Para realizar modificaciones de un documento en una colección, emplearemos el método **`findByIdAndUpdate`**. Buscará el documento con el *id* indicado, y reemplazará los campos atendiendo a los criterios que indiquemos como segundo parámetro.

En este enlace podéis consultar los operadores de actualización que podemos emplear en el segundo parámetro de llamada a este método. El más habitual de todos es `$set`, que recibe un objeto con los pares clave-valor que queremos modificar en el documento original. Por ejemplo, así reemplazamos el nombre y la edad de un contacto con un determinado *id*, dejando el teléfono sin modificar:

```
Contacto.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
{$set: {nombre:'Nacho Iborra', edad: 40}}, {new:true}).then(resultado => {
    console.log("Modificado contacto:", resultado);
}).catch (error => {
    console.log("ERROR:", error);
```

```
});
```

El tercer parámetro que recibe `findByIdAndUpdate` es un conjunto de opciones adicionales. Por ejemplo, la opción `new` que se ha usado en este ejemplo indica si queremos obtener como resultado el nuevo objeto modificado (`true`) o el antiguo antes de modificarse (`false`, algo útil para operaciones de deshacer).

Ahora ya puedes realizar el Ejercicio 7 del final de la sesión.

2.7.1. Actualizar la versión del documento

Hemos visto que, entre los atributos de un documento, además del *id* autogenerado por Mongo, se crea un número de versión en un atributo `__v`. Este número de versión alude a la versión del documento en sí, de forma que, si posteriormente se modifica (por ejemplo, con una llamada a `findByIdAndUpdate`), se pueda también indicar con un cambio de versión que ese documento ha sufrido cambios desde su versión original.

Si quisiéramos hacer eso con el ejemplo anterior, bastaría con añadir el operador `$inc` (junto al `$set` utilizado antes) para indicar que incremente el número de versión, por ejemplo, en una unidad:

```
Contacto.findByIdAndUpdate('5a0e1991075e9407c4da8b0a',
{$set: {nombre:'Nacho Iborra', edad: 40}, $inc: {__v: 1}}, {new:true}).then(...
```

Para finalizar, tienes propuesto el Ejercicio 8 al final de la sesión, de carácter optativo.

2.8. Más opciones de Mongoose

Lo visto hasta aquí es sólo una pequeña muestra de lo que ofrece la librería Mongoose, ya que para verla de forma más exhaustiva haría falta prácticamente un curso entero. Como muestra de lo que nos dejamos en el tintero, podemos citar estas variantes:

- La búsqueda parametrizada con find admite otras variantes de sintaxis, como el uso de métodos enlazados `where`, `limit`, `sort`... hasta obtener los resultados deseados en el orden y cantidad deseada. Por ejemplo, esta consulta muestra los 10 primeros contactos mayores de edad, ordenados de mayor a menor edad:

  ```
  Contacto.find().where('edad').gte(18).sort('-edad').limit(10).then(...
  ```

- Las actualizaciones admiten otros métodos aparte de los ya comentados. Así, se puede emplear el método `update`, en lugar de `findByIdAndUpdate`. En este caso, se actualizarán de golpe todos los documentos que cumplan las condiciones indicadas. Esta instrucción pone a 20 los años de todos los contactos llamados "Nacho":

  ```
  Contacto.update({nombre: 'Nacho', {$set: { edad: 20 }}).then(...
  ```

3. Ejercicios

3.1. Ejercicio 1

Como ya es habitual de sesiones anteriores, para los ejercicios de esta sesión crearemos una carpeta llamada "Sesion3" en nuestro espacio "ProyectosNode/Ejercicios", y dentro de esta carpeta añadiremos otra llamada "Ejercicio_3" donde iremos desarrollando los ejercicios que se proponen en este apartado. Verás que se trata de un ejercicio incremental, donde poco a poco iremos añadiendo código sobre un mismo proyecto.

Para empezar, instala Mongoose en dicho proyecto "Ejercicio_3" siguiendo los pasos indicados al principio del punto 2 de esta sesión, y crea un archivo "index.js" que, de momento, conectará con una base de datos llamada "libros", en el servidor Mongo local. Establece también el gestor de promesas por defecto de Javascript, como se explica también en ese apartado.

Recuerda que, aunque la base de datos aún no exista, no es problema para establecer una conexión, hasta que se añadan colecciones y documentos a ella.

3.2. Ejercicio 2

Sobre el "Ejercicio_3" iniciado anteriormente, vamos a definir un esquema para almacenar la información que nos interese de los libros. En concreto, almacenaremos su título, editorial y precio en euros. El título y el precio son obligatorios, el título debe tener una longitud mínima de 3 caracteres, y el precio debe ser positivo (mayor o igual que 0). Define estas reglas de validación en el esquema, y asócialo a un modelo llamado "libro" (con lo que se creará posteriormente la colección "libros" en la base de datos).

3.3. Ejercicio 3

Vamos a hacer un par de inserciones sobre la base de datos y modelo creados en los ejercicios anteriores. Bajo el código que ya deberás tener implementado (conectar con la base de datos y definir el modelo), haz lo siguiente:

- Crea un libro con estos datos:
  - Título: "El capitán Alatriste"
  - Editorial: "Alfaguara"
  - Precio: 15 euros
- Crea otro libro con estos otros datos:
  - Título: "El juego de Ender"
  - Editorial: "Ediciones B"
  - Precio: 8.95 euros

Inserta los dos libros en la base de datos. Deberán aparecer en la colección "libros". Al insertar, muestra por pantalla el resultado de la inserción, y si algo falla, muestra el error completo.

NOTA: si ejecutas la aplicación más de una vez, se añadirán los libros nuevamente a la colección, ya que no hemos puesto ninguna regla de validación para eliminar duplicados. No es problema. Siempre puedes eliminar los duplicados a mano desde Robo 3T.

3.4. Ejercicio 4 (opcional)

Intenta añadir a la colección un libro con estos datos:

- Título: "A"
- Editorial: vacía (no especificarla)
- Precio: 12 euros

La inserción deberá fallar (el título es demasiado corto). Captura el error que se produce, y muéstralo por consola.

3.5. Ejercicio 5

Sobre los libros que hemos insertado previamente, vamos a mostrar dos búsquedas:

- En primer lugar, utiliza el método genérico `find` para buscar los libros cuyo precio oscile entre los 10 y los 20 euros (inclusive)
- A continuación, utiliza `findById` para mostrar la información del libro que quieras (averigua el *id* de alguno de los libros y saca su información).

3.6. Ejercicio 6

Sobre la colección de libros anterior, localiza uno de los libros que deberías tener insertados de ejercicios anteriores. Quédate con su *id*, y bórralo de la colección empleando el método `findByIdAndRemove`. Muestra por pantalla los datos del libro borrado, cuando todo haya ido correctamente.

3.7. Ejercicio 7

Sobre la colección de libros anterior, localiza alguno de los libros que hayas insertado, quédate con su *id* y modifica su precio al valor que quieras. Muestra por pantalla los datos del nuevo libro modificado, una vez se haya completado la operación.

3.8. Ejercicio 8 (opcional)

Vamos a modificar ahora otro libro (o el mismo del ejercicio anterior, si quieres). En este caso, le volvemos a modificar el precio incrementándolo en 10 euros, y actualiza también la versión del documento (campo __v) incrementándola en una unidad.