3. Desarrollo de servicios con Node.js

Parte II. Acceso a bases de datos MongoDB con Mongoose

Programación de Servicios y Procesos

Nacho Iborra
Álvaro Pérez
Javier Carrasco

Tabla de contenido

6. Introducción a MongoDB....3
6.1. Introducción a JSON y BSON....3
6.1.1. Conversión desde y hacia JSON....3
6.1.2. Formato MongoDB y BSON....4
6.2. Instalación y configuración de MongoDB....4
6.2.1. Configurar una carpeta para las bases de datos....5
6.2.2. Iniciar el servidor....6
6.2.3. Instalación de MongoDB como servicio....6
6.3. Utilizar una GUI para acceder a MongoDB: Robo 3T....7
6.3.1. Descarga e instalación....7
6.3.2. Conectando al servidor....8
6.4. Algunos conceptos adicionales sobre bases de datos NoSQL....9
7. Uso de *"mongoose"*....10
7.1. Conectando al servidor....10
7.1.1. Configurar un *promise handler*....10
7.2. Modelos y esquemas....11
7.2.1. Definir el esquema....11
7.2.2. Aplicar el esquema a un modelo....11
7.2.3. Restricciones y validaciones....11
7.3. Insertar, eliminar, actualizar y buscar documentos....12
7.3.1. Insertar documentos....12
7.3.2. Sobre la identificación automática....14
7.3.3. Buscar documentos....14
7.3.4. Eliminar documentos....15
7.3.5. Actualizar de documentos....16
7.4. Combinar esquemas....17
7.4.1. Listado de colecciones con referencias a otras colecciones....18

6. Introducción a MongoDB

En esta parte se verá cómo conectarse a sistemas NoSQL. Este tipo de sistemas de bases de datos se ha vuelto muy popular recientemente y tratan los datos de una manera diferente a como lo hacen las bases de datos SQL. La principal diferencia es que están orientados a objetos, por lo que los datos se tratan como tal.

6.1. Introducción a JSON y BSON

JSON son las siglas de *JavaScript Object Notation*, una sintaxis de Javascript para representar objetos como cadenas. De esta manera, es muy fácil enviar este objeto a través de un *socket* o flujo, para ser almacenado en una base de datos remota o en un archivo de texto.

Un objeto Javascript se define como un conjunto de propiedades y valores. Por ejemplo, se pueden almacenar datos personales de esta manera:

```
let person = {
    name: "Javier",
    age: 44
};
```

Este mismo objeto se puede convertir a JSON, por lo que se obtendría una cadena como esta:

```
{"name": "Javier", "age": 44}
```

Si se dispone de una colección de objetos como esta:

```
let people = [
    { name: "Javier", age: 44},
    { name: "Mario", age: 4},
    { name: "Laura", age: 2},
    { name: "Nora", age: 10}
]
```

Si se transformara la colección a formato JSON, se obtendría la misma sintaxis, agrupada por corchetes:

```
[{"name":"Javier","age":44}, {"name":"Mario","age":4},
{"name":"Laura","age":2}, {"name":"Nora","age":10}]
```

6.1.1. Conversión desde y hacia JSON

Javascript proporciona un par de métodos útiles para transformar de Javascript a JSON y viceversa. Se continuará con la colección mostrada antes. Si se quiere transformar en JSON, se puede usar el método ***JSON.stringify***. Tiene un solo parámetro (el objeto o colección Javascript a transformar) y devuelve el formato JSON de estos datos.

```
let peopleJSON = JSON.stringify(people);
```

También se puede hacer lo contrario. Se puede utilizar el método ***JSON.parse***, que obtiene una cadena JSON y devuelve un objeto (o colección) Javascript extraído de esa cadena:

```
let people2 = JSON.parse(peopleJSON);
```

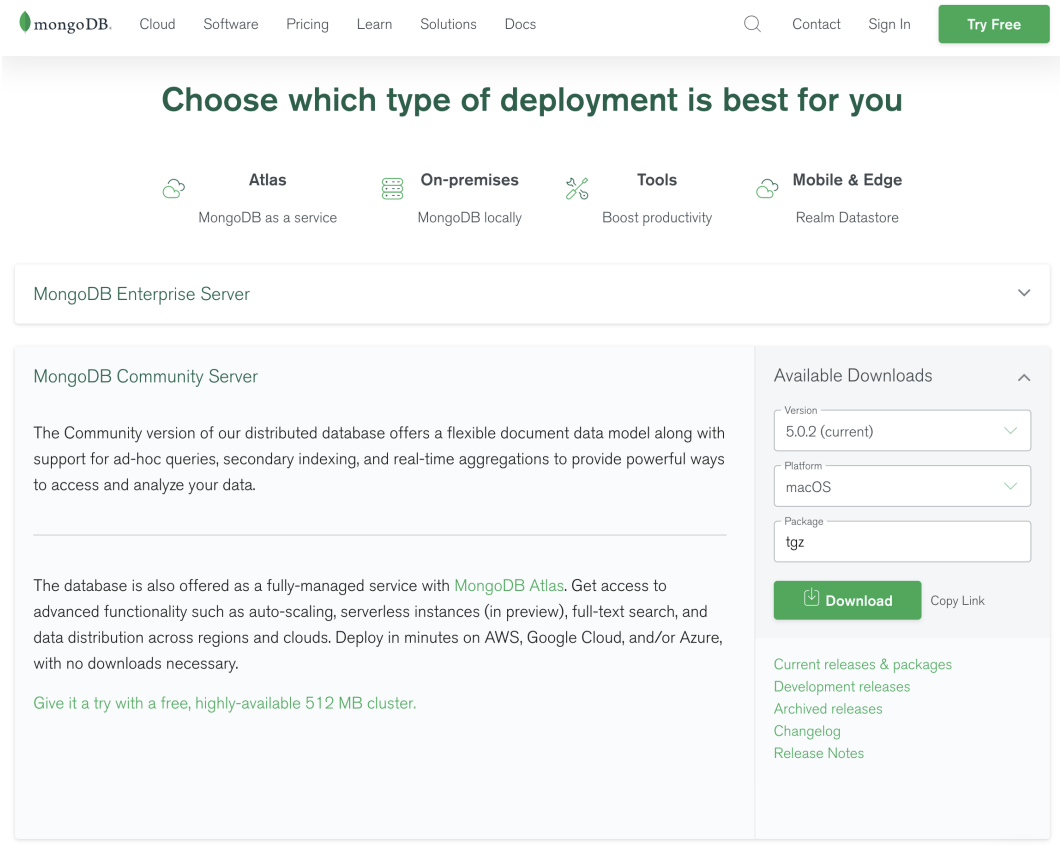
6.1.2. Formato MongoDB y BSON

Como se verá, MongoDB almacena la información utilizando un formato muy similar a JSON, llamado BSON, por lo que las bases de datos Mongo se pueden integrar fácilmente con aplicaciones Javascript (Node).

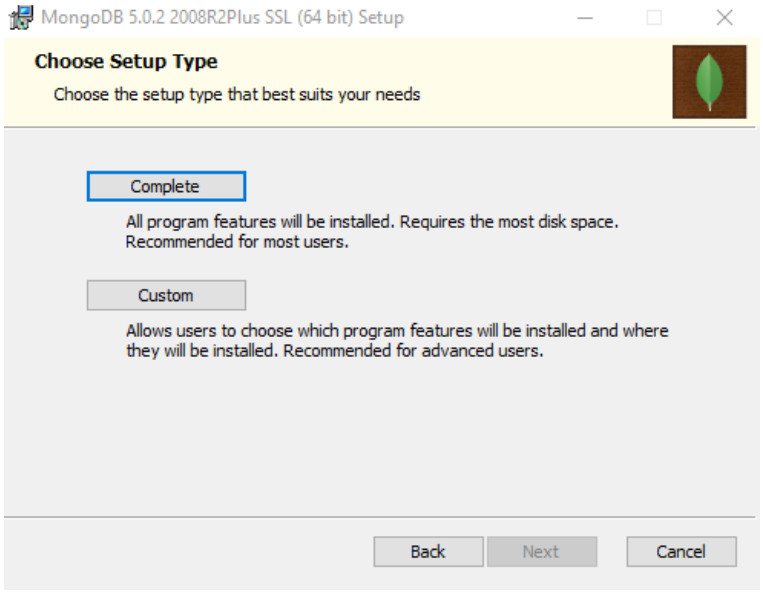
6.2. Instalación y configuración de MongoDB

MongoDB es un sistema de base de datos NoSQL, el más popular, de hecho. Es de código abierto y multiplataforma, por lo que se puede instalar en Mac, Windows o Linux.

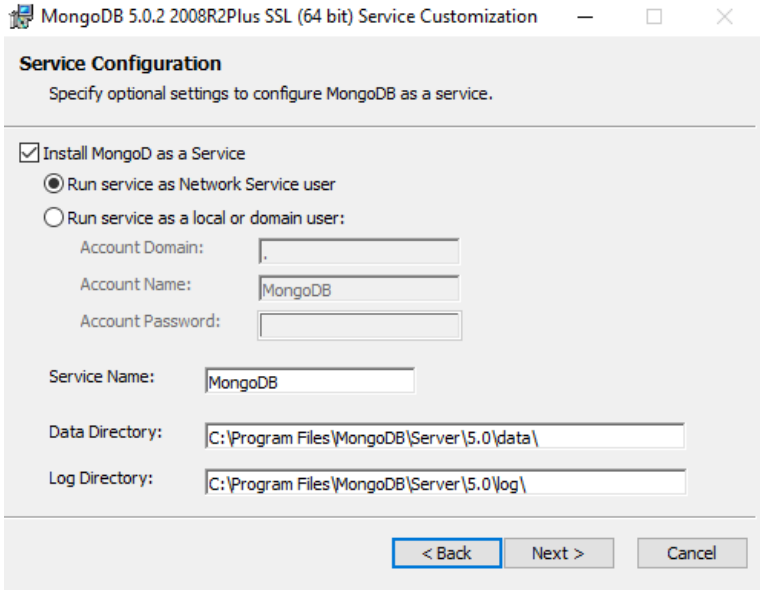
Si deseas descargar e instalar MongoDB, puedes ir a su página web oficial, a la sección *Software*. En ella, selecciona la opción *Community Server* y descarga la versión ***MongoDB Community Server*** para la plataforma deseada.



- Si utilizas un sistema Linux o Mac OS X, obtendrás un archivo *.tar.gz* que deberás descomprimir. La carpeta que obtendrás puede (debe) cambiarse de nombre y moverse donde quieras (por ejemplo, al directorio de inicio). Puedes llamar a esta carpeta “mongo”.
- Para los usuarios de Windows, se descargará un instalador *.msi*, que lanza un asistente para instalar MongoDB. Deberás elegir la instalación completa.



Presta atención a la carpeta donde está instalado MongoDB (normalmente *Archivos de programa\MongoDB\Server\XY*, siendo *XY* el número de versión). Habrá que ir a esta carpeta más tarde para iniciar el servidor.



6.2.1. Configurar una carpeta para las bases de datos

Si instalaste MongoDB en Linux o Mac OS X, necesitarás definir una carpeta para almacenar las bases de datos que creamos. Normalmente, esta carpeta se establece en nuestra carpeta de inicio, con el nombre “mongo-data” (aunque puedes elegir cualquier otra ubicación y nombre de carpeta). Si realizaste la instalación en Windows, esta carpeta se encontrará en la instalación, en el directorio “data”.

6.2.2. Iniciar el servidor

Para iniciar el servidor, una vez que esté instalado y la carpeta de la base de datos creada, deberás ir a la sub-carpeta *bin* dentro de la instalación de MongoDB y ejecutar el comando *mongod*, usando el parámetro *--dbpath* para indicar la ruta a la carpeta de la base de datos. Por ejemplo, en Mac se escribiría algo como esto para el usuario *javier*:

```
./mongod ––dbpath /Users/javier/mongo–data
```

Respecto a Windows, deberás ejecutar el comando *mongod.exe* desde el terminal, con los mismos parámetros.

Después de ejecutar el comando, el servidor estará esperando conexiones, escuchando en su puerto predeterminado (27017). Siempre que desees finalizar y cerrar el servidor, simplemente escribe *Control + C* en el terminal para finalizar el proceso.

Si utilizas una máquina virtual o un equipo diferente, deberás editar el fichero *mongod.cfg* que encontrarás en el directorio *bin* dentro de la instalación y modificar el parámetro *bindIP*, cambiado el valor *localhost* por *0.0.0.0* o la dirección IP del equipo.

6.2.3. Instalación de MongoDB como servicio

También se puede instalar MongoDB como servicio, especialmente en sistemas Linux, con estos comandos:

```
sudo apt–get update
sudo apt–get install mongodb
```

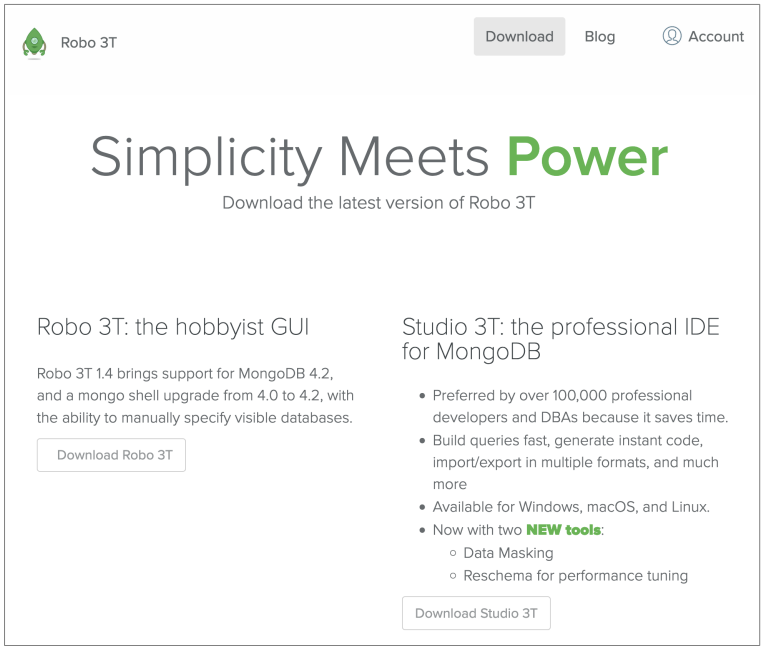
Después, se puede iniciar, detener o reiniciar el servidor con estos comandos:

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

Sin embargo, para esta unidad te recomendamos utilizar el primer modo de instalación, ya que no se necesita que el servicio esté funcionando continuamente en el sistema.

6.3. Utilizar una GUI para acceder a MongoDB: Robo 3T

Se puede utilizar el terminal para conectarse a la base de datos y enviar algunos comandos para crear bases de datos y añadir datos, pero esta opción es muy tediosa. En lugar de esto, se utilizará una GUI (interfaz gráfica de usuario) para facilitar estas tareas. El elegido es **Robo 3T** (antes llamado *RoboMongo*). Es un administrador gratuito y multiplataforma, que se puede descargar desde su página web oficial.



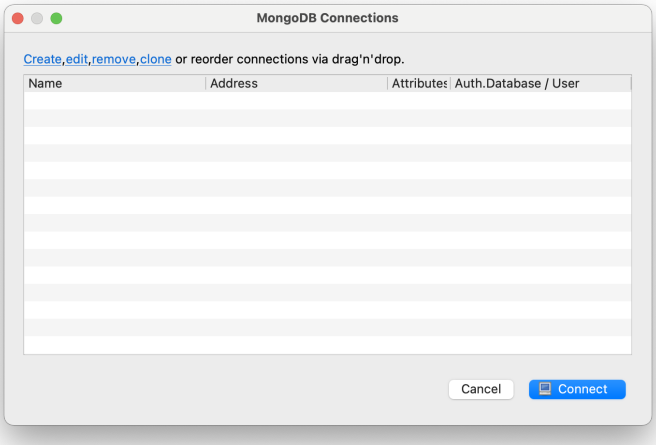
6.3.1. Descarga e instalación

Necesitarás descargar **Robo 3T**, que es la versión gratuita. Haz clic en el botón *Download Robo 3T*, y después podrás elegir la opción correspondiente a tu sistema operativo:

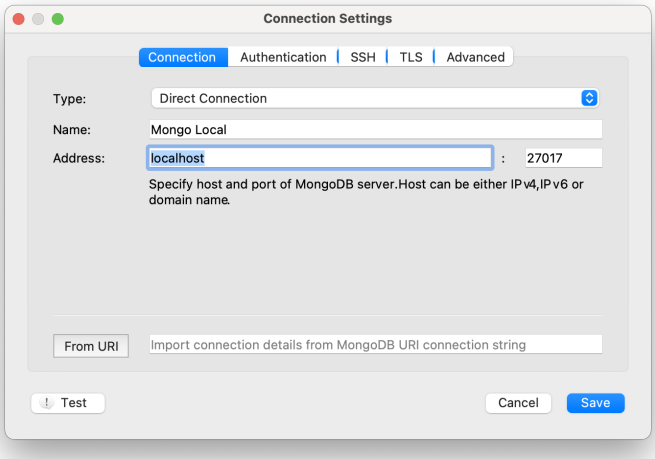
Respecto a Mac, se descargará un asistente para simplemente arrastrar la aplicación a la carpeta Aplicaciones. En Windows, se obtiene un asistente paso a paso que instala automáticamente Robo 3T en el sistema. Para los usuarios de Linux, se obtendrá un archivo portátil *.tar.gz*, que después deberá descomprimirse y ejecutar la aplicación desde la subcarpeta *bin*.

6.3.2. Conectando al servidor

Una vez que instalado Robo 3T, la primera vez que lo ejecutes verás un panel de conexión vacío:

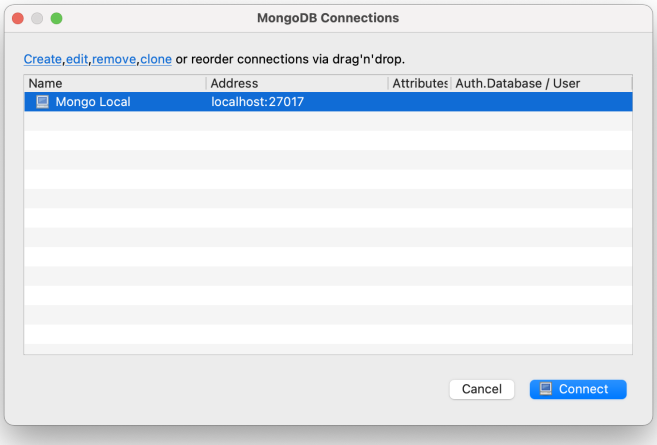


Aquí deberá especificarse la conexión al servidor (asumiendo que el servidor Mongo se está ejecutando en este momento). Haz clic en el enlace *Create* en la esquina superior izquierda, y especifica los parámetros de conexión:

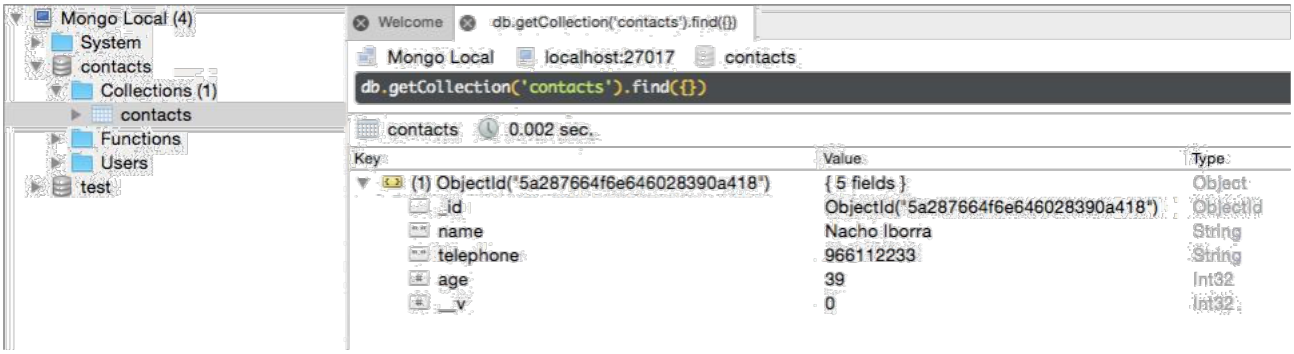


- Nombre de la conexión, por ejemplo, “Mongo Local”.
- La dirección y el puerto del servidor suelen ser correctos (localhost y 27017).

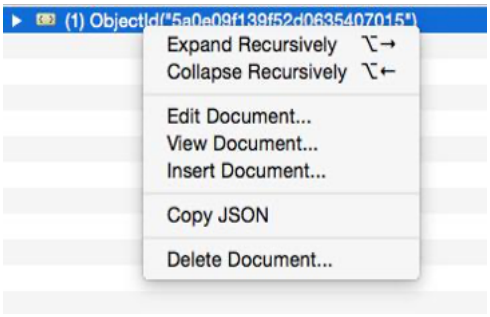
Guarda los cambios, y tendrás esta conexión disponible en el panel de conexiones. Seleccionala haz clic en el botón *Connect*.



Cuando se conecte, verás otra ventana. A la izquierda podrás ver todas las bases de datos existentes en este servidor. Puedes hacer clic derecho sobre cualquier base de datos para ver su contenido en la parte derecha:



Además, si haces clic derecho en cualquier registro de la derecha, podrás editarlo/eliminarlo/visualizarlo:



6.4. Algunos conceptos adicionales sobre bases de datos NoSQL

Las bases de datos NoSQL y SQL tienen algunas similitudes y diferencias. Ambas trabajan con **bases de datos**, es decir, lo que se crea en ambos casos es una base de datos. Pero en las bases de datos SQL, la información está estructurada en tablas, que tienen una lista de registros, mientras que en las bases de datos NoSQL la información está estructurada en **colecciones**, que tienen una lista de **documentos** dentro de ellas.

Se trabajará con colecciones, compuestas por documentos. Cada documento consta de un objeto (almacenado en formato BSON) con unas propiedades o campos, similares a los campos que puedes encontrar en una tabla. Pero es posible que hayan documentos con diferentes campos dentro de una misma colección (aunque no es lo habitual). Esto no se puede hacer en una tabla SQL.

7. Uso de *"mongoose"*

Hay algunas bibliotecas disponibles en el repositorio de NPM para gestionar bases de datos de MongoDB, pero la más popular es ***Mongoose***. Permite acceder fácilmente a una base de datos MongoDB y, además, permite definir esquemas, una estructura de validación para determinar el tipo de datos de cada atributo de los objetos que se van gestionar, junto con algunas otras restricciones, como si son requeridos o no, valores mínimos o máximos, etc. Puedes consultar algunos ejemplos e información en su página web oficial[1]. Comienza instalando la biblioteca en los proyectos deseados. Para hacer esto, deberás definir un archivo *"package.json"* en el proyecto con el comando *npm init*, y luego instala *mongoose* con *npm install*.

```
npm init
npm install mongoose
```

Una vez que esté instalado, deberás solicitarlo (*require*) en cada archivo fuente donde se vaya a utilizar:

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

7.1. Conectando al servidor

Para conectarse al servidor Mongo (siempre que se inicie con el comando *mongod*), necesitarás llamar al método de conexión desde el objeto mangosta. Obtiene la URL de la base de datos como parámetro; por ejemplo, si quieres conectarte a una base de datos llamada *"contacts"*, deberás escribir algo como esto:

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

No te preocupes si esta base de datos no existe en el servidor. Se creará automáticamente tan pronto como se inserten datos en ella.

7.1.1. Configurar un *promise handler*

Además de la conexión, es necesario definir un parámetro adicional para establecer el tipo de "promesas" que *Mongoose* utilizará. Una promesa es un tipo especial de objeto cuyo contenido no se establece de inmediato, se establece de forma asincrónica más tarde. Una vez completada la promesa, se podrá ejecutar un fragmento de código y continuar. Hay varios objetos de promesa implementados en las bibliotecas de Javascript, y habrá que decirle a *Mongoose* cuál se usará. Si solo se utiliza el objeto *Promise* predeterminado de Javascript, añade simplemente esta línea antes o después de conectar a la base de datos.

1 *Mongoose* (https://mongoosejs.com/)

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

7.2. Modelos y esquemas

Antes de comenzar a añadir y mostrar documentos de la base de datos, es necesario definir la estructura que va a tener cada documento. Como se ha dicho antes, los documentos almacenan objetos, por lo que se definirán las propiedades o atributos de cada objeto. Para ello se definirá un esquema y se asociará a un modelo (una colección en la base de datos).

7.2.1. Definir el esquema

Para definir un esquema, será necesario crear una instancia de la clase ***Schema*** que viene con *Mongoose*. Se crea el objeto y se definen los atributos que tendrá, junto con el tipo de datos de cada atributo. Por ejemplo, para cada contacto de la base de datos, será necesario su nombre, número de teléfono y edad:

```
let contactSchema = new mongoose.Schema({
    name: String,
    telephone: String,
    age: Number
});
```

Los tipos de datos disponibles para usar en los esquemas son:

- Textos (*String*).
- Números (*Number*).
- Fechas (*Date*).
- Booleanos (*Boolean*).
- Arrays (*Array*) .
- Otros: *Buffer*, *Mixed*, *ObjectId*.

7.2.2. Aplicar el esquema a un modelo

Una vez que definido el esquema, debe aplicarse a una colección determinada dentro de la base de datos. Para hacer esto, se utiliza el método ***model*** del objeto *mongoose*. Como primer parámetro, se indicará el nombre de la colección en la base de datos a la que se aplicará el esquema, y como segundo parámetro el esquema a aplicar (el que se definió antes, por ejemplo):

```
let Contact = mongoose.model('contacts', contactSchema);
```

7.2.3. Restricciones y validaciones

Si se define un esquema simple como el del ejemplo anterior, se permite que el usuario añada documentos con, por ejemplo, nombres vacíos o edades negativas. Es necesario proporcionar algunas instrucciones de validación para que los objetos con datos no válidos

se descarten automáticamente. Existen algunos validadores útiles que se pueden usar en los esquemas de Mongo (puedes ver una explicación detallada aquí). Por ejemplo, se puede establecer si un atributo es obligatorio o no con el validador ***required***, o un valor predeterminado con ***default***, una longitud mínima y/o máxima de una cadena con ***minlength*** y ***maxlength***, y algunas otras características, como expresiones regulares, identificadores únicos…

Con respecto al esquema de contactos, se establecerá que el nombre y el número de teléfono sean necesarios, y solo se permitirán edades entre 18 y 120 años (inclusive). Además, el nombre deberá tener una longitud mínima de 1 carácter y el número de teléfono estará compuesto por 9 dígitos. También se pueden eliminar los espacios en blanco al principio o al final de las cadenas con el validador ***trim***. Añade todas estas restricciones al esquema de esta manera:

```
let contactSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    telephone: {
        type: String,
        required: true,
        trim: true,
        match: /^\d{9}$/
    },
    age: {
        type: Number,
        min: 18,
        max: 120
    }
});
```

En cuanto a las expresiones regulares, puedes echar un vistazo a esta página si quieres obtener más información sobre qué tipo de símbolos se permiten dentro de ellas.

7.3. Insertar, eliminar, actualizar y buscar documentos

Ahora que ya tienes la conexión a la base de datos y el esquema de la colección de contactos, se comenzará a lanzar algunas operaciones.

7.3.1. Insertar documentos

Si quieres insertar o añadir un nuevo documento a una colección, se creará un objeto del modelo correspondiente y se llamará al método ***save***. Este método devuelve una promesa, por lo que se usará un bloque de código ***then*** para colocar las instrucciones si la operación ha sido correcta, y el bloque *catch* para establecer qué hacer si ocurre un error. Por ejemplo, a continuación, se añade un nuevo contacto con datos válidos.

```
let contact1 = new Contact({
    name: "Javier",
    telephone: "966112233",
```

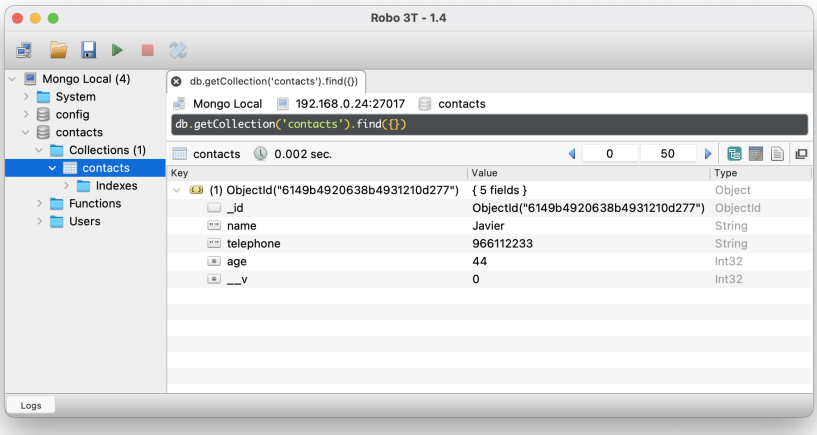
```
    age: 44
});

contact1.save().then(result => {
    console.log("Contact added:", result);
}).catch(error => {
    console.log("ERROR adding contact:", error);
});
```

Echa un vistazo al objeto resultado que se obtiene si todo está bien:

```
{
  name: 'Javier',
  telephone: '966112233',
  age: 44,
  _id: new ObjectId("6149b4920638b4931210d277"),
  __v: 0
}
```

Observa que se obtienen los mismos atributos que se añadieron anteriormente (nombre, número de teléfono y edad), y dos atributos adicionales que no se especificaron: uno llamado *__v* que se refiere a la versión del esquema (no se tratará este atributo en esta unidad) y otro llamado *_id* que es un identificador automático generado por MongoDB. Se explorará este identificador con más profundidad en unas pocas líneas. Pero ahora, ve a Robo 3T y examina la bases de datos nuevamente. Tendrás una base de datos de *"contacts"* creada en el panel izquierdo, y si haces clic derecho sobre la colección de *"contacts"* para ver sus documentos, podrás ver el nuevo contacto añadido:



Si intentas insertar un objeto no válido, se saltará a la sección ***catch***. Por ejemplo, este contacto es demasiado antiguo según las restricciones del esquema:

```
let contact2 = new Contact({
    name: "Matuzalem",
    telephone: "965123456",
    age: 200
});
contact2.save().then(result => {
    console.log("Contact added:", result);
}).catch(error => {
    console.log("ERROR adding contact:", error);
});
```

Si echas un vistazo al mensaje de error, verás un atributo ***ValidationError*** con esta información:

```
ERROR adding contact: Error: contacts validation failed: age: Path `age` (200) is more than
maximum allowed value (120).
```

7.3.2. Sobre la identificación automática

Si has insertado correctamente los documentos anteriores en la colección, habrás notado que hay una propiedad llamada ***_id*** con un código generado automáticamente. Notarás que este no es un valor entero, sino un tipo de cadena. De hecho, es un texto de 12 bytes que almacena información sobre:

- La marca de tiempo de creación del documento, es decir, la fecha y hora exactas en que se creó el documento.
- El equipo que creó el documento. Esto es particularmente útil cuando se escala la aplicación y existen varios servidores Mongo accediendo a la misma base de datos.
- El proceso que creó el documento.
- Un contador aleatorio (por si acaso todos los parámetros anteriores son iguales).

Existen algunos métodos para extraer parte de esta información, como la marca de tiempo de creación, pero no se tratará de momento.

De todos modos, también se puede establecer una identificación propia, pero no es una buena opción:

```
let contactX = new Contact({_id:2, name:"Juan", telephone:"611885599"});
```

7.3.3. Buscar documentos

Si se desea buscar cualquier documento (o lista de documentos) en una colección, se puede usar la método ***find***. Si no especificamos ningún parámetro, obtendremos toda la colección en un promesa:

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

También se puede filtrar los documentos que coincidan con un criterio de búsqueda determinado, como *id*, nombre, número de teléfono o una combinación de algunos de ellos. Estos criterios de filtro se pasan como un parámetro al método *find*.

```
Contact.find({name: 'Javier', telephone: '966112233'}).then(result => {
    console.log("Contact found:", result);
}).catch (error => {
    console.log("ERROR:", error);
});
```

Hay otras opciones para buscar documentos en una colección, como el método ***findOne*** (para obtener solo un resultado que coincida con los criterios del filtro), o ***findById*** (que filtra por *id* exclusivamente).

> **NOTA**: si la consulta es correcta, pero no encuentra ningún resultado, entonces no se obtendrá ningún error (es decir, no saltará a la sección *catch*). Se obtendrá un *array* vacío (si se utiliza *find*), o nulo (si se usa *findOne* o *findById*).

7.3.4. Eliminar documentos

Para borrar o eliminar un documento de una colección, se puede utilizar estos métodos estáticos:

- ***deleteMany*** o ***deleteOne***, elimina TODOS, o uno, los documentos que coinciden con los criterios de filtro pasados como parámetro. Si no se establece ningún parámetro, se eliminarán todos los elementos de la colección.

```
Contact.deleteMany({_id: '5a16fed09ed79f03e490a648'}).then(result => {
    console.log("Contact removed:", result);
}).catch (error => {
    console.log("ERROR:", error);
});
```

  Observa que *result* contiene el número de elementos que se han borrado.

- ***findOneAndRemove***, encuentra un documento que coincide con el filtro especificado (solo un documento) y lo elimina. Además, se obtendrá el documento eliminado en el resultado, por lo que incluso se puede deshacer la operación si se desea (añadiendo el documento nuevamente, manualmente).

```
Contact.findOneAndRemove({_id: '5a16fed09ed79f03e490a648'}).then(result => {
    console.log("Contact removed:", result);
}).catch (error => {
    console.log("ERROR:", error);
});
```

  Ten en cuenta que, en este caso, *result* es el documento que se ha eliminado.

- ***findByIdAndRemove***, busca un documento por su *id* y lo elimina. Este también obtiene el documento eliminado como resultado.

```
Contact.findByIdAndRemove('5a16fed09ed79f03e490a648').then(result => {
    console.log("Contact removed:", result);
}).catch (error => {
    console.log("ERROR:", error);
});
```

7.3.5. Actualizar de documentos

Por último, si se desea actualizar los atributos de un documento, se puede utilizar el método ***findByIdAndUpdate***. Buscará el documento con el *id* especificado y reemplazará los atributos que se especifiquen en la llamada al método. Este método tiene tres parámetros:

- La identificación a buscar, como se hace en el método *findByIdAndRemove*, por ejemplo.
- Los operadores de actualización y los atributos afectados. Puedes echar un vistazo a esta página para ver qué operadores se pueden utilizar. El más típico es ***$set***, para cambiar el valor antiguo de un atributo por uno nuevo. Pero también puedes utilizar otros operadores, como ***$inc*** (simplemente aumenta el valor de un atributo), o ***$unset*** (quita el valor), entre otras opciones.
- Opciones adicionales. En este último parámetro, se podrá especificar una opción llamada ***new*** para determinar si se quiere recuperar el nuevo documento actualizado (*true*) o el anterior (*false*, valor predeterminado).

Aquí puedes ver un ejemplo: en este caso, se actualiza el nombre y la edad de un contacto con una identificación determinada y le se le dice a Mongo que devuelva el documento nuevo y actualizado como resultado.

```
Contact.findByIdAndUpdate('6149b5f3e82a15bd09129e7c',
    {$set: {name:'Javi', age: 39}},
    {new:true}).then(result => {
        console.log("Contact updated:", result);
    }).catch (error => {
        console.log("ERROR:", error);
    });
```

Ejercicio 3

Crea un proyecto llamado ***"Exercise_ContactsMongo"*** en tu espacio de trabajo. Usa los comandos *npm init* y *npm install* para instalar *mongoose* en el proyecto y crea un archivo fuente llamado ***"contacts.js"***. Añade el código explicado anteriormente para conectarse a una base de datos llamada ***"contacts"*** y define el esquema que se muestra arriba para el modelo *Contact*. Después, deberás implementar varias acciones dependiendo de un parámetro adicional que utilizarás al iniciar la aplicación:

- Si llamas a la aplicación así: ***node contacts.js i***, deberás añadir 3 contactos válidos a la colección *"contacts"*. Estos contactos estarán predefinidos en código, no se solicitan al usuario.
- Si utilizas ***node contacts.js l*** se mostrarán dos listas:
  - Listado de todos los contactos de la colección.
  - Muestra un contacto filtrando por su nombre y teléfono.

- Si utilizas ***node contacts.js u***, deberás actualizar el nombre de un contacto, dado su *id* (o cualquier otro atributo, si lo deseas).
- Finalmente, si usas ***node contacts.js d***, debes eliminar un contacto por su *id*.

Ten en cuenta que ninguna de estas opciones debe preguntarle nada al usuario. Simplemente añadirán, eliminarán o eliminarán documentos predefinidos. Por tanto, debes ajustar el código para especificar la acción de eliminar o actualizar, por ejemplo.

Para obtener este parámetro adicional (***i***, ***l***, ***u*** o ***d***), puedes usar ***process.argv***, que obtiene la lista completa de argumentos. El primer argumento (*process.argv[0]*) es el comando *node*, y el segundo (*process.argv[1]*) es el archivo fuente a ejecutar (*"contacts.js"*, en este caso). Por tanto, debes acceder a *process.argv[2]* para determinar que operación se debe realizar.

7.4. Combinar esquemas

Una de las opciones más poderosas de *Mongoose* y MongoDB es que se pueden definir esquemas que son parte de otros esquemas. Por ejemplo, se podría definir un esquema de dirección para almacenar la información de la dirección de un contacto dado (calle, número, ciudad, código postal) y luego añadir un nuevo atributo a nuestro esquema de contacto para especificar esta dirección. Estos dos esquemas y modelos podrían ser como estos:

```
let addressSchema = new mongoose.Schema({
    street: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    number: {
        type: Number,
        required: true,
        min: 1
    },
    city: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    postalCode: {
        type: Number,
        minlength: 1
    }
});

let Address = mongoose.model('addresses', addressSchema);

let contactSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    telephone: {
        type: String,
```

```
        required: true,
        trim: true,
        match: /^\d{9}$/
    },
    age: {
        type: Number,
        min: 18,
        max: 120
    },
    address: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'addresses',
        required: true,
    }
});

let Contact = mongoose.model('contacts', contactSchema);
```

Después, si quieres crear un nuevo contacto, necesitaras crear y guardar la dirección correspondiente previamente (si aún no existe en la base de datos), y luego guardar el nuevo contacto:

```
let address = new Address({
    street: "C/La piedra verde", number: 128,
    city: "Alicante", postalCode: 3005
});

address.save().then(resultAddress => {
    let contact = new Contact({
        name: "Javier Carrasco",
        telephone: "966112233",
        age: 39,
        address: "614da8cf7a93c2d7bd7d23c5"
    });

    contact.save().then(resultContact => {
        console.log("Contact added:");
        console.log(resultContact);
    }).catch(error => {
        console.log("Error adding contact:", error);
    });
}).catch(error => {
    console.log("Error adding address:", error);
});
```

7.4.1. Listado de colecciones con referencias a otras colecciones

¿Qué pasa si se desea obtener una lista de todos los contactos incluyendo toda la información sobre sus direcciones? Se podría pensar que es necesario lanzar una llamada a ***find*** para obtener todos los contactos, y luego, para cada contacto, definir una llamada *find* para obtener las direcciones correspondientes de cada contacto… pero esto se puede hacer con solo una llamada *find*. Se puede utilizar el método ***populate*** para indicar que se necesita completar la información con respecto a un atributo en particular (la dirección en este caso). Echa un vistazo a las siguientes instrucciones y los resultados que se obtienen, respectivamente:

Opción 1: búsqueda simple

```
Contact.findById("61520e69ebc3f4d5423f0c45").then(result => {
    console.log(result);
});
```

Resultado:

```
{
  _id: new ObjectId("61520e69ebc3f4d5423f0c45"),
  name: 'Javier Carrasco',
  telephone: '966112233',
  age: 39,
  address: new ObjectId("614da8cf7a93c2d7bd7d23c5"),
  __v: 0
}
```

Opción 2: búsqueda con *populate*

```
Contact.findById("61520e69ebc3f4d5423f0c45").populate('address').then(result => {
    console.log(result);
});
```

Resultado:

```
{
  _id: new ObjectId("61520e69ebc3f4d5423f0c45"),
  name: 'Javier Carrasco',
  telephone: '966112233',
  age: 39,
  address: {
    _id: new ObjectId("614da8cf7a93c2d7bd7d23c5"),
    street: 'C/La piedra verde',
    number: 128,
    city: 'Alicante',
    postalCode: 3005,
    __v: 0
  },
  __v: 0
}
```

El método *populate* se puede encadenar, de modo que se puedan poblar varios atributos, si tienes múltiples referencias a múltiples colecciones:

```
Collection.findById("61520e69ebc3f4d5423f0c45").populate('attribute1')
    .populate('attribute2')...
    .then(result => { console.log(result); });
```

Ejercicio 4

Crea un nuevo proyecto llamado ***"Exercise_ContactsMongoAddress"*** en tu espacio de trabajo. Como antes, instala *mongoose* y crea un archivo fuente *"contactos.js"*. En este caso, se pedirá que definas ambos esquemas en el archivo de origen (*address* y *contact*) y luego:

- Si ejecutas la aplicación con el parámetro ***i***, se añade un nuevo contacto con su dirección correspondiente (ambos predefinidos en código).
- Si ejecutas la aplicación con el parámetro ***l***, se listará todos los contactos poblando el atributo *"address"*.

Ejercicio 5

Crea un proyecto llamado ***"Exercise_Books"*** en tu espacio de trabajo. Instala *mongoose* y crea un archivo llamado *"books.js"*. Conéctate a una base de datos llamada *"books"* y define estos dos esquemas, con sus modelos correspondientes:

- Esquema para el modelo *"Author"*:
  - *firstName*: *String*, obligatorio, min. longitud de 1, con *trim*.
  - *lastName*: *String*, obligatorio, min. longitud de 1, con *trim*.
  - *yearOfBirth*: Número, no obligatorio, valores entre 1900 y 2000 (inclusive).
- Esquema del modelo *"Book"*:
  - *title*: *String*, obligatorio, min. longitud de 1, con *trim*.
  - *author*: una referencia al modelo *"Author"*, obligatorio.
  - *price*: Número, no obligatorio, pero debe ser positivo.

Una vez definidos los esquemas y modelos, sigue los siguientes pasos:

- Si se llama a la aplicación con el parámetro ***i***, se añadirán 3 libros con sus autores correspondientes (al menos, se debe repetir un autor).
- Si se llama a la aplicación con el parámetro ***l***, se listarán todos los libros correspondientes a una identificación de autor determinada, poblando el atributo de autor.
- Si se llama a la aplicación con el parámetro ***u***, se actualizará la información de un autor: cambiar su año de nacimiento.