educación IT
CODE YOUR FUTURE

Bootcamp
Full Stack Engineer

Fase 3 - Desarrollo Back End

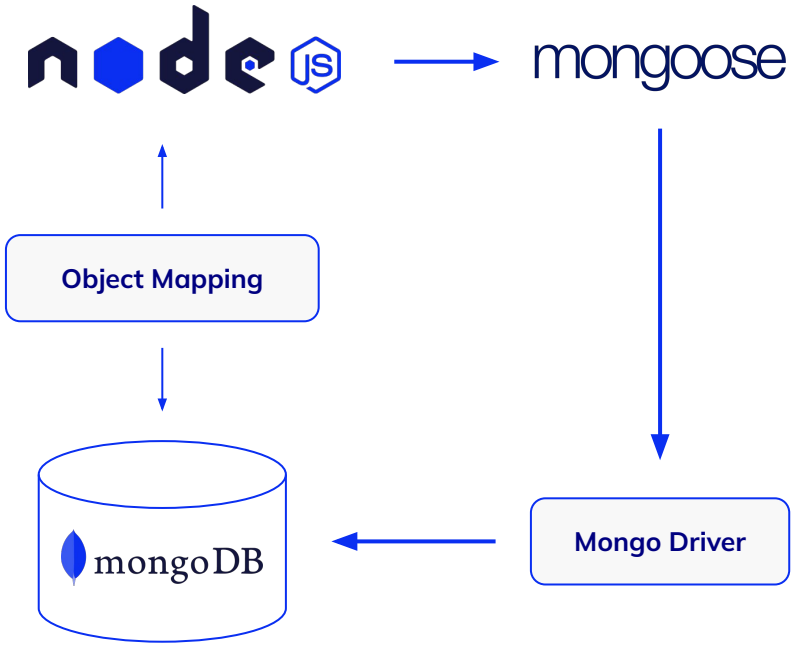
Módulo 35

Mongoose para MongoDB y Node.js

Introducción

Mongoose es un marco de JavaScript usado, en general, **en una aplicación Node.js con una base de datos MongoDB**.

A continuación, presentaremos Mongoose y MongoDB y conoceremos dónde se adaptan estas tecnologías para su aplicación.



¿Qué es MongoDB?

MongoDB es una base de datos que **almacena sus datos como documentos**. Lo más común es que estos sean similares a una estructura JSON:

```
{
  firstName: "Jamie",
  lastName: "Munro"
}
```

Luego, un documento se coloca **dentro de una colección**. El ejemplo del documento anterior define un objeto **`User`**. Este normalmente sería parte de una colección llamada **`Users`**.

Uno de los factores más importantes con MongoDB es **su flexibilidad** cuando se trata de estructura. Aunque en el primer ejemplo, el objeto **`user`** contenía una propiedad **`firstName`** y **`lastName`**, estas no son necesarias en todos los documentos **`user`** que forman parte de la colección **`users`**. Esto es lo que hace a **MongoDB** ser muy diferente de una base de datos SQL como MySQL o Microsoft SQL Server, las cuales requieren un esquema de base de datos fuertemente definido de cada objeto que almacena.

La capacidad de crear **objetos dinámicos que se almacenan como documentos en la base de datos** es donde **Mongoose** entra en juego.

¿Qué es Mongoose?

Es un ***Object Document Mapper*** **(ODM)**. Esto significa que **permite definir objetos con un esquema fuertemente tipado que se asigna a un documento MongoDB**.

Con Mongoose, contamos con muchas funcionalidades que nos permiten **crear y trabajar con esquemas**. Al momento, contiene **ocho SchemaTypes** que una propiedad se guarda como cuando se conserva a MongoDB:

1. **`String`** (Cadena)
2. **`Number`** (Número)
3. **`Date`** (Fecha)
4. **`Buffer`**
5. **`Boolean`** (Booleano)
6. **`Mixed`** (Mixto)
7. **`ObjectId`**
8. **`Array`** (Matriz).

mongoose

Cada tipo de datos posibilita especificar:

- Un valor **predeterminado**.
- Una función de **validación personalizada**.
- Que se requiere **un campo**.
- Una función `get` que le permite manipular los datos **antes de que se devuelva como un objeto**.
- Una función de conjunto que le permite manipular los datos **antes de guardarlos en la base de datos**.
- Crear índices para permitir que los datos se obtengan **más rápido**.

Además de estas opciones, ciertos tipos de datos permiten personalizar aún más **la manera en que se almacenan y recuperan los datos de la base**. Por ejemplo, con un tipo de datos `String` podemos especificar estas opciones adicionales:

- Convertirlo a **minúsculas**.
- Convertirlo a **mayúsculas**.
- **Recortar datos antes de guardar**.
- Una expresión regular que puede **limitar los datos que se pueden guardar durante el proceso de validación**.
- Una enumeración que puede definir **una lista de cadenas que son válidas**.

Las propiedades `Número` (*Number*) y `Fecha` (*Date*) son compatibles con la especificación de un valor **mínimo y máximo** permitido para ese campo. La mayoría de los ocho tipos de datos permitidos deberían serte familiares, pero puede haber varias excepciones, como **`Buffer`, `Mixed`, `ObjectId` y `Array`**.

`Buffer` permite **guardar datos binarios**. Un ejemplo habitual de este tipo de datos es **una imagen** o **un archivo codificado**, como por ejemplo **un documento PDF**.

`Mixed` convierte la propiedad en un campo "todo vale". representa un tipo sin esquema, que puede cambiar el valor a cualquier otra cosa que desee porque no hay una estructura definida.

Tenga cuidado con su uso ya que **pierde muchas de las funciones que ofrece Mongoose**, como la validación de datos y la detección de cambios de entidades para saber automáticamente si desea actualizar la propiedad al guardar.

`ObjectId` especifica un **enlace a otro documento en su base de datos**. Por ejemplo, si tiene una colección de libros y autores, el documento del libro puede contener una propiedad `ObjectId` que hace referencia al autor del documento.

Con **`Array`** podemos **almacenar matrices similares a JavaScript**. Con este tipo de datos, haremos operaciones de matriz JavaScript comunes en ellos: `push`, `pop`, `shift`, `slice`, etc.

Recapitulación rápida

- **MongoDB** es **una base de datos que permite almacenar documentos con una estructura dinámica**. Estos documentos se guardan **dentro de una colección**.
- **Mongoose** es **una biblioteca de JavaScript que permite definir esquemas con datos fuertemente tipados**. Al definir un esquema, Mongoose permite crear un modelo basado **en un esquema específico**. Un modelo de Mongoose se asigna a un documento MongoDB **a través de la definición del esquema del modelo**.
- Tras estar definidos sus modelos y esquemas, Mongoose contiene funciones para **validar, guardar, eliminar y consultar sus datos usando las funciones comunes de MongoDB**.

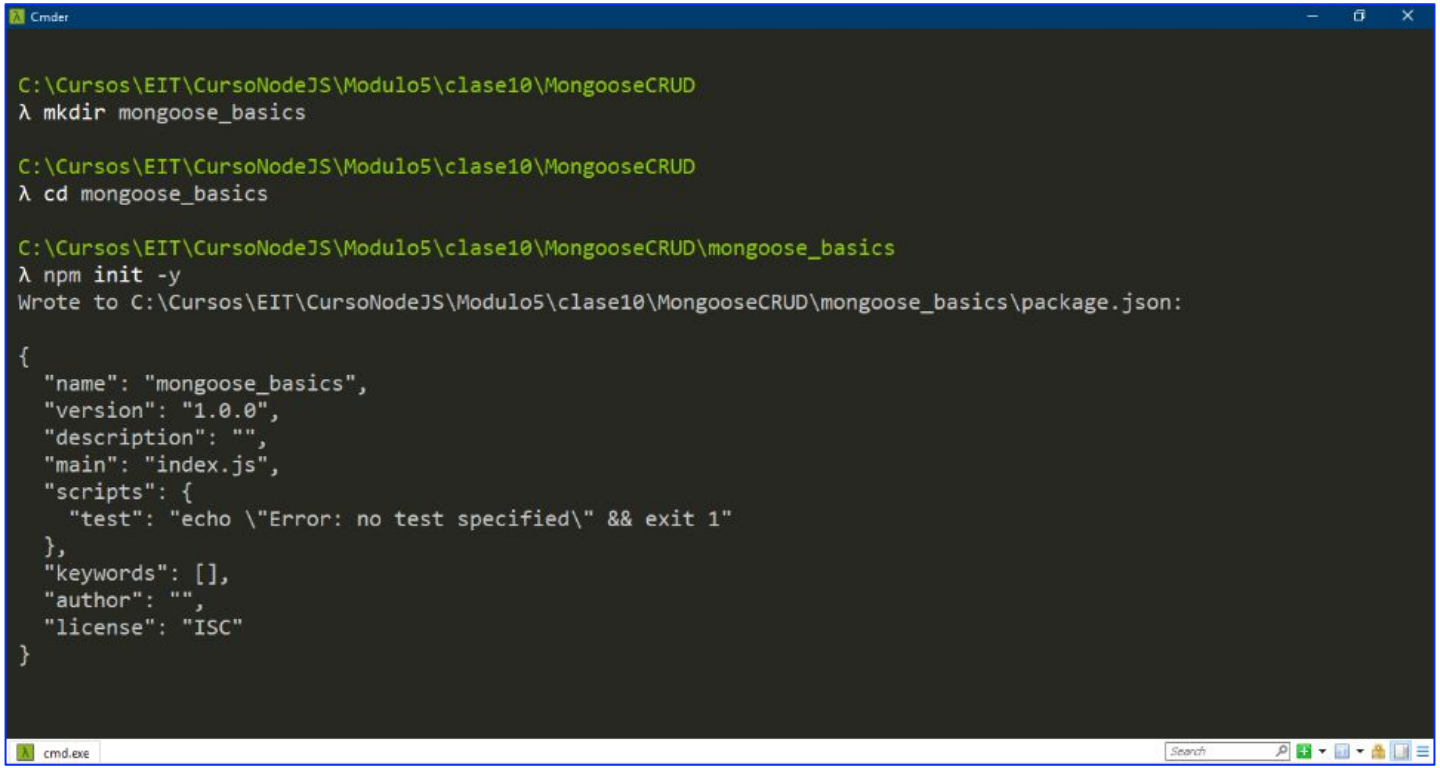
Configuración de Mongoose

Mongoose es un marco de JavaScript y, por lo tanto, lo usaremos **en una aplicación Node.js**. Si ya tenemos Node.js instalado, pasamos al paso que sigue, de lo contrario, **visitar la página de descarga de Node.js y elegir el instalador para nuestro sistema operativo**.

1. Con Node.js configurado y listo, crearemos una **nueva aplicación** y luego instalaremos **el paquete Mongoose NPM**.

2. Con un símbolo del sistema que está configurado donde desea instalar su aplicación, puede ejecutar **los siguientes comandos**:

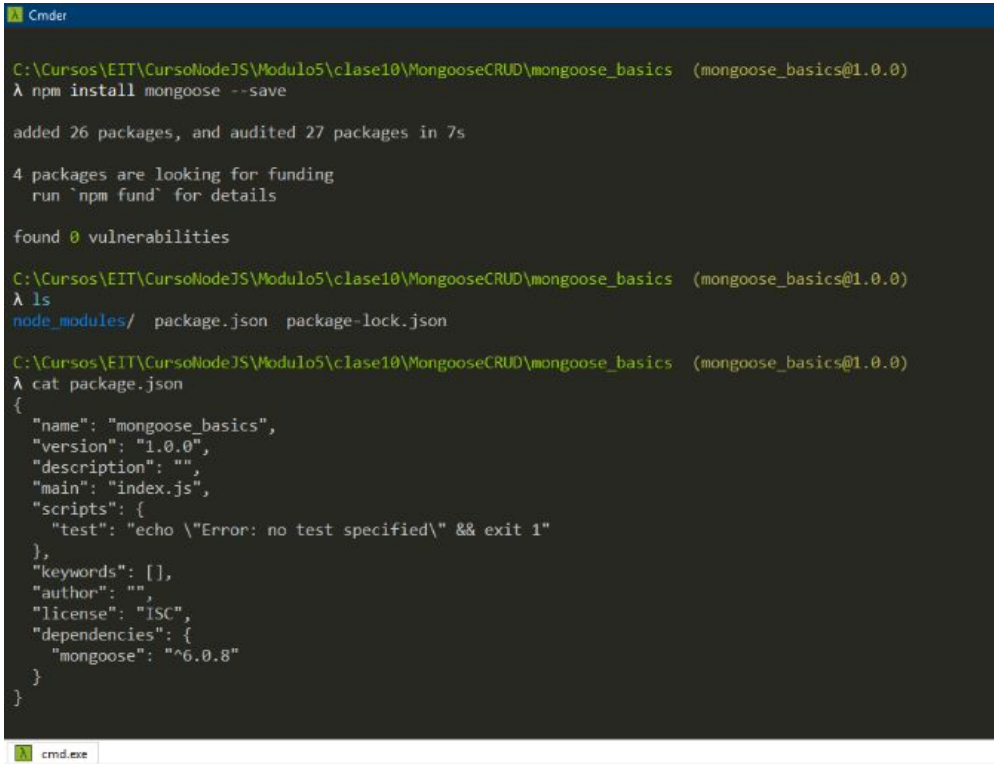
```
mkdir mongoose_basics
cd mongoose_basics
npm init -y
```


Cmder
C:\Cursos\EIT\CursoNodeJS\Modulo5\clase10\MongooseCRUD
λ mkdir mongoose_basics
C:\Cursos\EIT\CursoNodeJS\Modulo5\clase10\MongooseCRUD
λ cd mongoose_basics
C:\Cursos\EIT\CursoNodeJS\Modulo5\clase10\MongooseCRUD\mongoose_basics
λ npm init -y
Wrote to C:\Cursos\EIT\CursoNodeJS\Modulo5\clase10\MongooseCRUD\mongoose_basics\package.json:
{
"name": "mongoose_basics",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
"test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [],
"author": "",
"license": "ISC"
}
cmd.exe
Search

3. Para la inicialización de mi aplicación, dejé sus **valores predeterminados**.

   Ahora instalaré el paquete de Mongoose de la siguiente manera:

```
npm install mongoose –save
```



4. Con todos los requisitos previos configurados, conectarse a una base de datos **MongoDB**. Yo coloqué el siguiente código dentro de un archivo **index.js** porque lo elegí como punto de partida para mi aplicación:

La primera línea de código incluye la biblioteca Mongoose. A continuación, abro una conexión a una base de datos a la que he llamado **mongoose_basics** utilizando la función **connect**.

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

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

Esta función acepta otros **dos parámetros opcionales**. El segundo es un objeto de opciones donde definiremos **datos como el nombre de usuario y la contraseña**, de ser preciso. El tercer parámetro, que también puede ser el segundo de no tener opciones, es **la función de devolución de llamada luego de intentar conectarse**. La función de devolución de llamada se puede usar **de una de estas dos formas**:

```
mongoose.connect(uri, options, function(error) {
    // Check error in initial connection. There is no 2nd param to the callback.
});

// Or using promises
mongoose.connect(uri, options).then(
    () => { /** ready to use. The `mongoose.connect()` promise resolves to undefined. */ },
    err => { /** handle initial connection error */ }
);
```

5. Para evitar una posible introducción a las promesas de JavaScript, **usaré la primera**. A continuación se muestra un archivo ***index.js*** actualizado:

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

mongoose.connect('mongodb://localhost/mong
oose_basics', function (err) {

   if (err) throw err;

   console.log('Successfully connected');

});
```

Si se produce un error al conectarse a la base de datos, **se lanza la excepción y se detiene todo el procesamiento posterior**. Registré un mensaje de éxito en caso de **no haber un error**.

Ahora, Mongoose está configurado y conectado a una base de datos llamada ***mongoose_basics***. Mi conexión MongoDB **no usa nombre de usuario, contraseña o puerto personalizado**. Si necesita establecer estas opciones o cualquier otra durante la conexión, **revise la documentación de Mongoose al conectarse**. En ella, encontrará explicaciones de las opciones disponibles, como también datos sobre cómo crear conexiones múltiples y agrupar conexiones, entre otras cosas.

Definiendo un esquema de Mongoose

Durante la introducción, mostré un objeto **user** que contenía dos propiedades: **firstName** y **lastName**. En el ejemplo que sigue, traducimos ese documento a un esquema de Mongoose:

```
var userSchema = mongoose.Schema({
    firstName: String,
    lastName: String
});
```

En él hay solo **dos propiedades sin atributos asociados**, por lo que ampliaremos el ejemplo transformando las propiedades del nombre y apellido **en objetos secundarios de una propiedad name**. Esta comprenderá ambos datos (tanto el nombre como el apellido). También agregaré una propiedad **created** de tipo **Date:**

```
var userSchema = mongoose.Schema({
    name: {
      firstName: String,
    lastName: String
    },
    created: Date
});
```

Como se ve, con Mongoose puedo construir **esquemas flexibles** con distintas combinaciones de cómo organizar mis datos.

A continuación, crearemos dos nuevos esquemas que demostrarán **cómo crear una relación con otro esquema: author y book**. **Book** contendrá una referencia al esquema **author**.

En la imagen de la derecha, vemos que en la parte de arriba tenemos el esquema `author` el cual **amplía los conceptos del esquema user creados en el ejemplo anterior**.

```
var authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
            firstName: String,
        lastName: String
    },
    biography: String,
    twitter: String,
    facebook: String,
    linkedin: String,
    profilePicture: Buffer,
    created: {
        type: Date,
        default: Date.now
    }
});
```

Si deseamos vincular **al autor y el libro**, la primera propiedad del esquema `author` es una propiedad `_id` que es un tipo de esquema **`ObjectId`**. **`_id`** es **la sintaxis común para crear una clave primaria en Mongoose y MongoDB**.

Luego, al igual que el esquema `user`, definimos una propiedad `name` que contiene **el nombre y apellido del autor**.

Ampliando el esquema `user`, `author` contiene varios otros tipos de esquema `String`. También añadimos un tipo de esquema de Buffer que podría contener la imagen de perfil del autor.

La propiedad final contiene la fecha de creación del autor, aunque se crea de manera distinta porque ha definido un valor predeterminado de "ahora". Cuando un autor persiste en la base de datos, esta propiedad se establecerá en la fecha / hora actual.

Para completar los ejemplos de esquema, creamos un esquema book que contenga **una referencia al autor utilizando el tipo de esquema `ObjectId`**:

```
var bookSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    title: String,
    summary: String,
    isbn: String,
    thumbnail: Buffer,
    author: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Author'
    },
```

...

```
    ratings: [
        {
            summary: String,
            detail: String,
            numberOfStars: Number,
            created: {
                type: Date,
                default: Date.now
            }
        }
    ],
    created: {
        type: Date,
        default: Date.now
    }
});
```

El esquema `book` contiene varias propiedades de tipo `String`. Como se mencionó antes, contiene una referencia al esquema `author`. Para demostrar aún más las potentes definiciones de esquema, book también contiene **un array de ratings**. Cada calificación tiene un **`summary`, `detail`, `numberOfStars` y una propiedad `created`**.

Con Mongoose podemos **crear esquemas con referencias a otros esquemas** o, como en el ejemplo anterior con la `ratings`, podemos crear un array de propiedades secundarias que podría estar contenida **en un esquema relacionado** (como libro a autor) **o en línea como en el ejemplo anterior** (con libro a un array de calificaciones).

Crear y guardar modelos de Mongoose

Ya que `author` y `book` evidencian la flexibilidad del esquema de Mongoose, continuaremos utilizándolos y derivaremos un modelo `Author` y `Book` de ellos:

```
var Author = mongoose.model('Author', authorSchema);
var Book = mongoose.model('Book', bookSchema);
```

Al guardar un modelo de Mongoose, crea un documento en MongoDB **con las propiedades definidas por el esquema del que se deriva**.

En el próximo ejemplo, veremos cómo se crea y guarda un objeto y, para eso, crearemos varios objetos: un modelo de **`Author`** y varios modelos **`Book`**. Una vez creados, estos se conservarán en MongoDB con el método `save` del modelo:

```
var jamieAuthor = new Author ({
    _id: new mongoose.Types.ObjectId(),
    name: {
        firstName: 'Jamie',
        lastName: 'Munro'
    },
    biography: 'Jamie is the author of ASP.NET MVC 5 with Bootstrap and Knockout.js.',
    twitter: 'https://twitter.com/endyourif',
    facebook: 'https://www.facebook.com/End-Your-If-194251957252562/'
});
```

...

```
jamieAuthor.save(function(err) {
    if (err) throw err;
    console.log('Author successfully saved.');

    var mvcBook = new Book ({
            _id: new mongoose.Types.ObjectId(),
            title: 'ASP.NET MVC 5 with Bootstrap and Knockout.js',
            author: jamieAuthor._id,
            ratings:[{
                summary: 'Great read'
            }]
    });

    mvcBook.save(function(err) {
        if (err) throw err;

        console.log('Book successfully saved.');
    });
```

```
    var knockoutBook = new Book ({
            _id: new mongoose.Types.ObjectId(),
            title: 'Knockout.js: Building Dynamic Client-Side Web Applications',
            author: jamieAuthor._id
    });

    knockoutBook.save(function(err) {
        if (err) throw err;

        console.log('Book successfully saved.');
    });
});
```

En el ejemplo anterior, ocultamos la referencia a **los libros más recientes**. El ejemplo comienza creando y guardando un **`jamieObject`** que se crea a partir de un modelo **`author`**. Dentro de la función `save` del `jamieObject`, si ocurre un error, la aplicación generará **una excepción**.

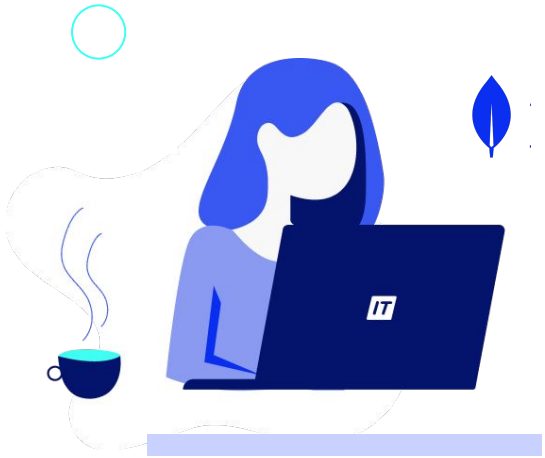
Cuando el guardado es exitoso, dentro de la función `save`, los dos objetos del libro se crean y guardan. De forma similar al **`jamieObject`**, de haber un error al guardar, se genera un **mensaje de error**, caso contrario, veremos **un mensaje de éxito en la consola**.

Para crear la referencia al autor, los objetos del libro hacen referencia a la clave primaria **`_id`** del esquema `author` en la propiedad `author` del esquema `book`.

Validación de datos antes de guardar

En general, los datos que terminan creando un modelo se llenan por un formulario en una página web. Por eso, se recomienda **validar los datos antes de guardar el Modelo en MongoDB**.

A continuación, actualicé el esquema de autor anterior y añadí una validación de las propiedades **`firstName`**, **`twitter`**, **`facebook`** y **`linkedin`**. Veamos la siguiente pantalla.



```
var authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
        firstName: {
            type: String,
            required: true
        },
        lastName: String
    },
    biography: String,
    twitter: {
        type: String,
        validate: {
            validator: function(text) {
                return text.indexOf('https://twitter.com/') === 0;
            },
            message: 'Twitter handle must start with https://twitter.com/'
        }
    },
```

...

```
facebook: {
    type: String,
    validate: {
        validator: function(text) {
            return text.indexOf('https://www.facebook.com/') === 0;
        },
        message: 'Facebook must start with https://www.facebook.com/'
    }
},
linkedin: {
    type: String,
    validate: {
        validator: function(text) {
            return text.indexOf('https://www.linkedin.com/') === 0;
        },
        message: 'LinkedIn must start with https://www.linkedin.com/'
    }
},
```

```
...

    profilePicture: Buffer,
    created: {
        type: Date,
        default: Date.now
    }
});
```

La propiedad **`firstName`** fue atribuida a la propiedad **`required`**. Ahora,al llamar a la función `save`, Mongoose devolverá **un error** con un mensaje que indica que precisamos la propiedad **`firstName`**. No hicimos requerida la propiedad **`lastName`** por aquellos nombres en mi base de datos que están compuestos solo por el nombre, como Cher o Madonna.

Las propiedades **`twitter`**, **`facebook`**, y **`linkedin`** tienen validadores personalizados muy similares que se les aplican. Cada uno de ellos garantiza que los valores comiencen con el nombre de dominio respectivo de las redes sociales. Estos campos no son necesarios, por lo que el validador solo se aplicará cuando se proporcionen datos para esa propiedad.

Búsqueda y actualización de datos

Una introducción a Mongoose no estaría completa sin aprender **cómo buscar un registro y actualizar una o más propiedades en ese objeto**. Mongoose proporciona varias funciones diferentes para encontrar datos para un modelo específico: **`find`**, **`findOne`**, y **`findById`**.

1. `Find` y `findOne` **aceptan un objeto como entrada para búsquedas complejas**, mientras que `findById` **acepta solo un valor con una función de devolución de llamada**. En la imagen de la derecha, veremos cómo encontrar todos los libros que contienen la cadena **`"mvc"`** en su título:

```
Book.find({
    title: /mvc/i
}).exec(function(err, books) {
    if (err) throw err;

    console.log(books);
});
```

Dentro de la función `find`, buscamos la cadena *"mvc"* que **no distingue entre mayúsculas y minúsculas en la propiedad `title`**. Logramos esto **utilizando la misma sintaxis para buscar una cadena con JavaScript**.

La función de búsqueda también se puede encadenar a otros métodos de consulta, **`where`, `and`, `or`, `limit`, `sort`, `any`**, etc.

2. Ampliemos el ejemplo de la imagen anterior y limitemos los resultados a **los primeros cinco libros**. Además, **se ordenarán de acuerdo a su fecha de creación en orden descendente**. Esto devolverá **los cinco libros más recientes que contengan "mvc" en el título**:

```
Book.find({
    title: /mvc/i
}).sort('-created')
.limit(5)
.exec(function(err, books) {
    if (err) throw err;

    console.log(books);
});
```

3. Tras aplicar la función **`find`**, el orden de las otras funciones no es importante ya que todas las funciones encadenadas **se compilan juntas en una sola consulta y no se ejecutan hasta que se llama a la función exec**.

4. Como vimos, **`findById`** se ejecuta ligeramente distinto: de forma inmediata y acepta una función de devolución de llamada en lugar de permitir una cadena de funciones. A continuación, veremos una consulta a un autor específico por su **`_id`**:

```
Author.findById('59b31406beefa1082819e72f',
function(err, author) {
    if (err) throw err;

    console.log(author);
});
```

En tu caso, el `_id` en puede ser diferente. Copié este `_id` de un **`console.log`** anterior al encontrar una lista de libros con "mvc" en su título.

5. Una vez que un objeto ha sido devuelto, **puede modificar cualquiera de sus propiedades para actualizarlo**. Tras realizar los cambios necesarios debe llamar al método `save`, de la misma manera que mientras creaba el objeto. Como vemos en la imagen, extenderé el ejemplo de **`findbyId`** y actualizaré la propiedad `linkedin` sobre el autor.

6. Tras recuperar el autor con éxito, se establece la propiedad **`linkedin`** y se invoca la función `save`. Mongoose puede detectar que `linkedin` se modificó y **enviará una declaración de actualización a MongoDB solo sobre las propiedades que se han modificado**. De haber un error al guardar, se lanzará **una excepción y detendrá la aplicación**. De ser exitosa, **se registra un mensaje de éxito en la consola**.

```
Author.findById('59b31406beefa1082819e72f',
function(err, author) {
    if (err) throw err;

    author.linkedin =
'https://www.linkedin.com/in/jamie-munro-8064ba1a/'
;

    author.save(function(err) {
        if (err) throw err;

        console.log('Author updated successfully');
    });
});
```

7. Mongoose cuenta con dos funciones adicionales que permiten **encontrar un objeto y guardarlo en un solo paso con las funciones apropiadamente nombradas**: **`findByIdAndUpdate`** y **`findOneAndUpdate`**. Actualicemos el ejemplo anterior para usar **`findByIdAndUpdate`**.

```
Author.findByIdAndUpdate('59b31406beefa1082819e72f'
,
    { linkedin:
'https://www.linkedin.com/in/jamie-munro-8064ba1a/'
},
    function(err, author) {
        if (err) throw err;

        console.log(author);
});
```

En este ejemplo, suministramos las propiedades para actualizar como un objeto para el segundo parámetro de la función **`findByIdAndUpdate`**. Ahora, la función de devolución de llamada es **el tercer parámetro**.

Si la actualización se realiza con éxito, entonces, el objeto `author` devuelto contendrá **la información actualizada**. Registramos esto en la consola para poder ver las propiedades del autor actualizado.

Código de muestra final

A lo largo de este manual, estudiamos pequeños fragmentos de código que identifican acciones concretas, como crear un esquema, crear un modelo, etc. A continuación, uniremos todo en un mismo ejemplo.

1. En primer lugar, creamos dos adicionales: **`author.js`** y **`book.js`**. Estos cuentan con sus respectivas **definiciones de esquema y la creación del modelo**. La última línea de código permite que el modelo esté disponible en el archivo **`index.js`**.

   Comencemos con el archivo `author.js`.

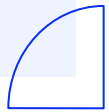
En la siguiente diapositiva, veremos la imagen del código.

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

var authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
        firstName: {
            type: String,
            required: true
        },
        lastName: String
    },
    biography: String,
    twitter: {
        type: String,
        validate: {
            validator: function(text) {
                return text.indexOf('https://twitter.com/') === 0;
            },
            message: 'Twitter handle must start with https://twitter.com/'
        }
    },
```

```
facebook: {
    type: String,
    validate: {
        validator: function(text) {
            return text.indexOf('https://www.facebook.com/') === 0;
        },
        message: 'Facebook must start with https://www.facebook.com/'
    }
},
linkedin: {
    type: String,
    validate: {
        validator: function(text) {
            return text.indexOf('https://www.linkedin.com/') === 0;
        },
        message: 'LinkedIn must start with https://www.linkedin.com/'
    }
},
```

...

```
    profilePicture: Buffer,
    created: {
        type: Date,
        default: Date.now
    }
});

var Author = mongoose.model('Author', authorSchema);
module.exports = Author;
```



2. Luego viene el archivo ***book.js***:

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

var bookSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    title: String,
    summary: String,
    isbn: String,
    thumbnail: Buffer,
    author: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Author'
    },
```

...

```
    ratings: [
        {
            summary: String,
            detail: String,
            numberOfStars: Number,
            created: {
                type: Date,
                default: Date.now
            }
        }
    ],
    created: {
        type: Date,
        default: Date.now
    }
});

var Book = mongoose.model('Book', bookSchema);

module.exports = Book;
```

3. Y finalmente, **el archivo *index.js* actualizado**:

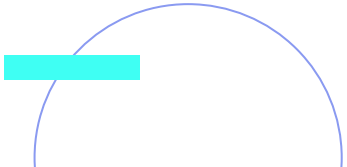
```
var mongoose = require('mongoose');

var Author = require('./author');
var Book = require('./book');

mongoose.connect('mongodb://localhost/mongoose_basics', function (err) {
    if (err) throw err;

    console.log('Successfully connected');
```

...

```
var jamieAuthor = new Author({
    _id: new mongoose.Types.ObjectId(),
    name: {
        firstName: 'Jamie',
        lastName: 'Munro'
    },
    biography: 'Jamie is the author of ASP.NET MVC 5 with Bootstrap and Knockout.js.',
    twitter: 'https://twitter.com/endyourif',
    facebook: 'https://www.facebook.com/End-Your-If-194251957252562/'
});
```

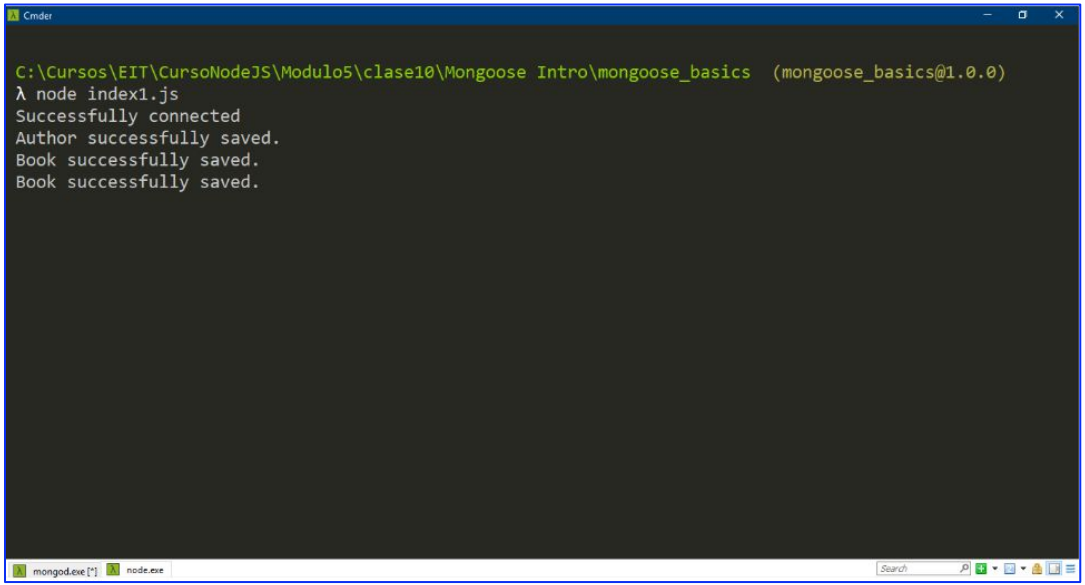
```
jamieAuthor.save(function(err) {
    if (err) throw err;

    console.log('Author successfully saved.');

    var mvcBook = new Book({
        _id: new mongoose.Types.ObjectId(),
        title: 'ASP.NET MVC 5 with Bootstrap and Knockout.js',
        author: jamieAuthor._id,
        ratings:[{
            summary: 'Great read'
        }]
    });

    mvcBook.save(function(err) {
        if (err) throw err;

        console.log('Book successfully saved.');
    });
```

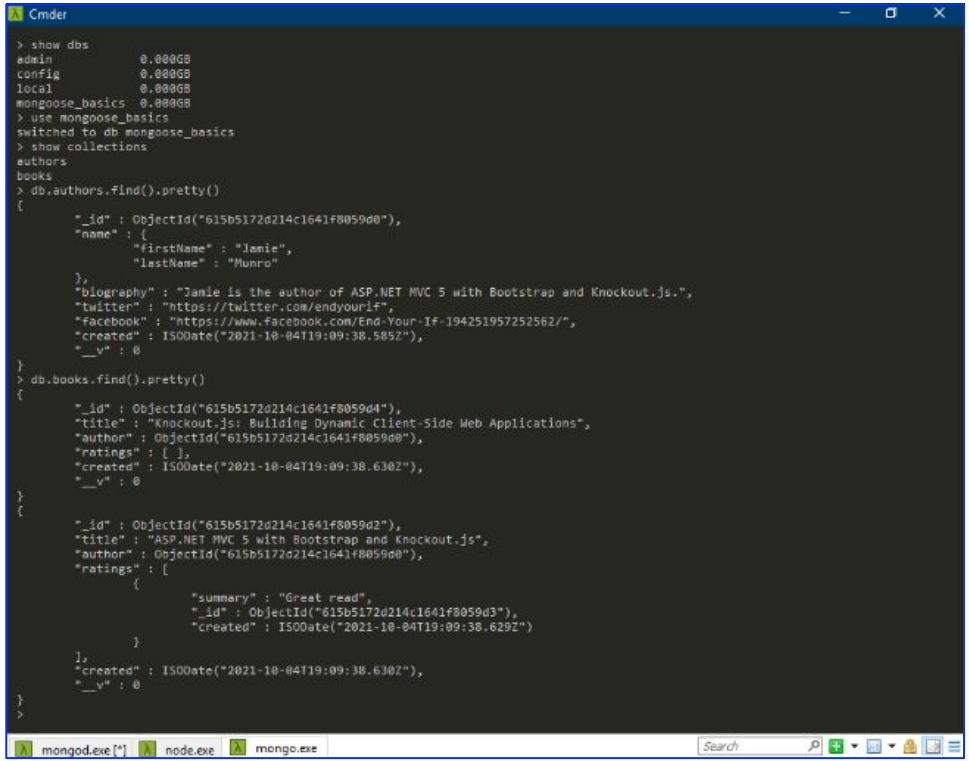
```
        var knockoutBook = new Book({
            _id: new mongoose.Types.ObjectId(),
            title: 'Knockout.js: Building Dynamic Client-Side Web Applications',
            author: jamieAuthor._id
        });

        knockoutBook.save(function(err) {
            if (err) throw err;

            console.log('Book successfully saved.');
        });
    });
});
```



Vimos que todas las acciones de Mongoose están dentro de la función **`connect`**. Incluimos los archivos `author` y `book` con la función `require` tras incluir la biblioteca Mongoose.

4. Con MongoDB en ejecución, ya podemos ejecutar la aplicación completa Node.js al usar el comando **`node index.js`**.





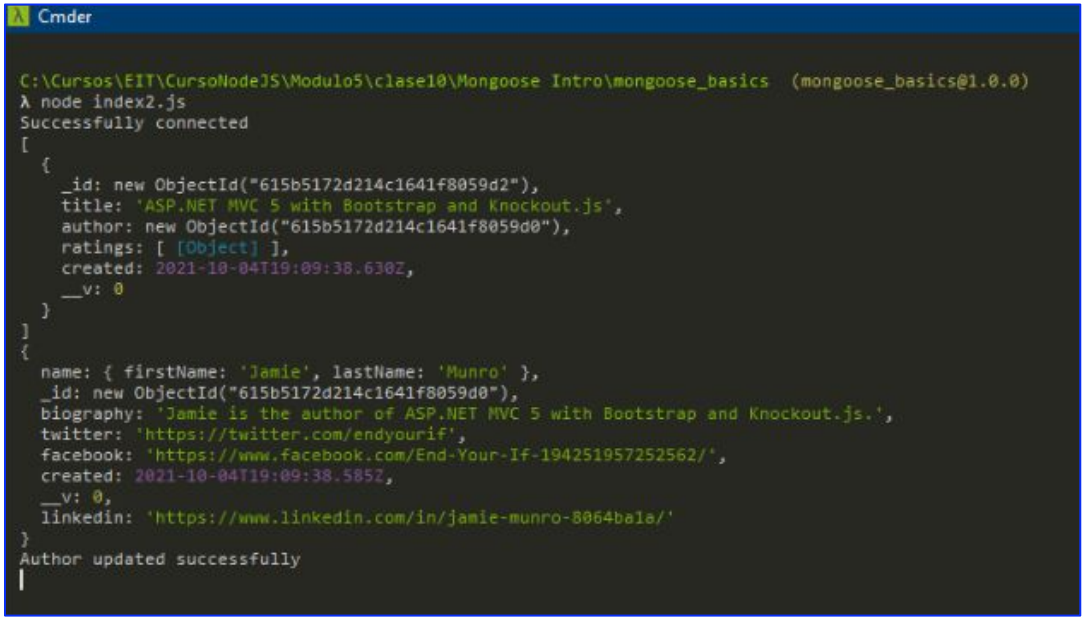
5. Tras guardar algunos datos en mi base, actualicé el archivo **`index.js`** con las funciones de búsqueda como se ve en la imagen de la siguiente diapositiva y su continuación.

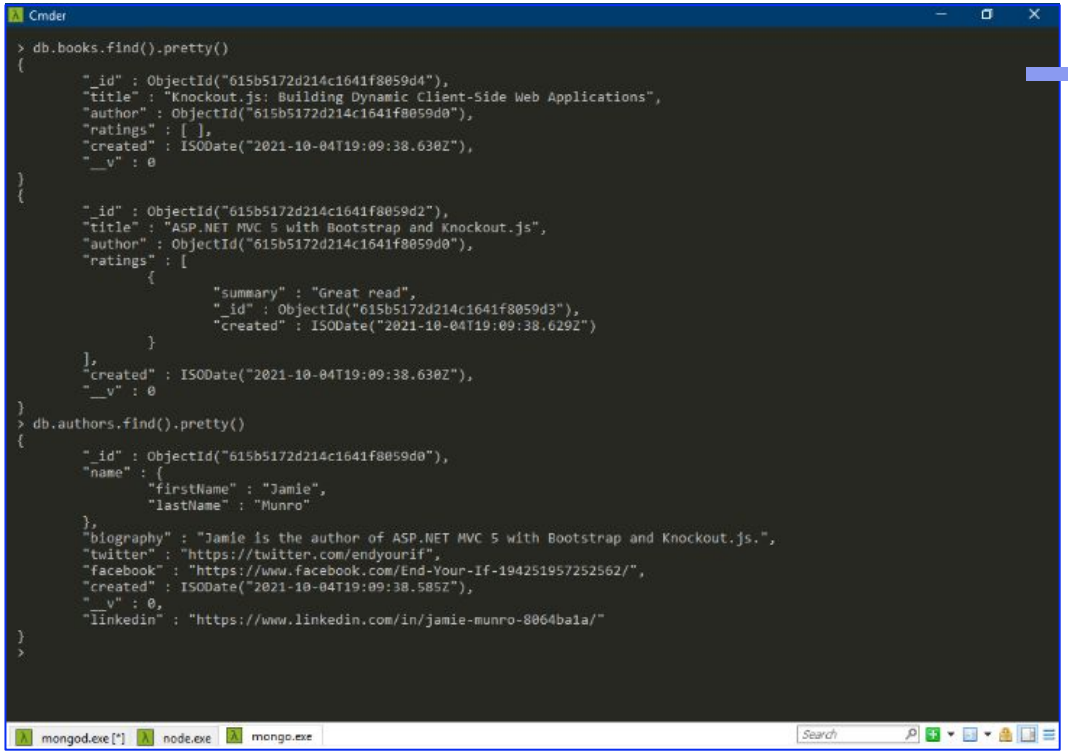
```
var mongoose = require('mongoose');

var Author = require('./author');
var Book = require('./book');

mongoose.connect('mongodb://localhost/mongoose_basics', function (err) {
    if (err) throw err;
    console.log('Successfully connected');

    Book.find({
        title: /mvc/i
    }).sort('-created')
    .limit(5)
    .exec(function(err, books) {
        if (err) throw err;

        console.log(books);
    });
```

...

```
    Author.findById('615b5172d214c1641f8059d0', function(err, author) {
        if (err) throw err;

        author.linkedin = 'https://www.linkedin.com/in/jamie-munro-8064ba1a/';

        author.save(function(err) {
            if (err) throw err;

            console.log('Author updated successfully');
        });
    });

    Author.findByIdAndUpdate('615b5172d214c1641f8059d0', { linkedin:
'https://www.linkedin.com/in/jamie-munro-8064ba1a/' }, function(err, author) {
        if (err) throw err;

        console.log(author);
    });
});
```

6. Una vez más, puede ejecutar la aplicación con el comando: **`node index.js`**.




Cmder
> db.books.find().pretty()
{
"_id" : ObjectId("615b5172d214c1641f8059d4"),
"title" : "Knockout.js: Building Dynamic Client-Side Web Applications",
"author" : ObjectId("615b5172d214c1641f8059d0"),
"ratings" : [ ],
"created" : ISODate("2021-10-04T19:09:38.630Z"),
"__v" : 0
}
{
"_id" : ObjectId("615b5172d214c1641f8059d2"),
"title" : "ASP.NET MVC 5 with Bootstrap and Knockout.js",
"author" : ObjectId("615b5172d214c1641f8059d0"),
"ratings" : [
{
"summary" : "Great read",
"_id" : ObjectId("615b5172d214c1641f8059d3"),
"created" : ISODate("2021-10-04T19:09:38.629Z")
}
],
"created" : ISODate("2021-10-04T19:09:38.630Z"),
"__v" : 0
}
> db.authors.find().pretty()
{
"_id" : ObjectId("615b5172d214c1641f8059d0"),
"name" : {
"firstName" : "Jamie",
"lastName" : "Munro"
},
"biography" : "Jamie is the author of ASP.NET MVC 5 with Bootstrap and Knockout.js.",
"twitter" : "https://twitter.com/endyourif",
"facebook" : "https://www.facebook.com/End-Your-If-194251957252562/",
"created" : ISODate("2021-10-04T19:09:38.585Z"),
"__v" : 0,
"linkedin" : "https://www.linkedin.com/in/jamie-munro-8064ba1a/"
}
>
mongod.exe [*]
node.exe
mongo.exe
Search

Resumen

Tras leer este manual, podrás **construir esquemas y modelos Mongoose extremadamente flexibles, aplicar validación simple o compleja, crear y actualizar documentos y buscar los documentos que se crearon**.

Espero que ahora te sientas cómodo usando Mongoose. Para obtener más información, revisa **las guías de Mangoose**, las cuales ahondan en temas avanzados como población, middleware y promesas, entre otros.

educación IT
CODE YOUR FUTURE

¡Sigamos trabajando!