3. Desarrollo de servicios con Node.js

Anexo II. Más opciones de Mongoose

Programación de Servicios y Procesos

Arturo Bernal
Nacho Iborra
Javier Carrasco

Tabla de contenido

1. Relaciones y subdocumentos

Se ha visto cómo definir una relación simple entre dos colecciones, de modo que un contacto tenga una dirección, o un libro tenga un autor. Pero existen otros tipos de relaciones entre colecciones, como las relaciones múltiples o subdocumentos.

1.1. Definir una relación múltiple

Se irá un paso más allá, se definirá una relación que permita vincular un documento de una colección con varios documentos de otra colección. Por ejemplo, se hará que los contactos tengan una lista de mascotas. Para ello, deberá definirse un nuevo esquema para las mascotas, con (por ejemplo) su nombre y tipo (perro, gato, …).

```
let petSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    type: {
        type: String,
        required: true,
        enum: ['dog', 'cat', 'other']
    }
});
let Pet = mongoose.model('pets', petSchema);
```

> Fíjate cómo usar el validador *enum* para forzar que un campo determinado tenga solo un conjunto limitado de valores.

Si se quiere permitir que un contacto tenga varias mascotas, deberá añadirse un nuevo campo al esquema, y se establecerá el tipo como ***Object arra***y, asociando el esquema mascotas definido.

```
let contactSchema = new mongoose.Schema({
    name: {
        …
    },
    telephone: {
        …
    },
    age: {
        …
    },
    address: {
        …
    },
    pets: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: 'pets'
    }]
});
let Contact = mongoose.model('contacts', contactSchema);
```

1.1.1. Insertar documentos con múltiples relaciones

Si se quiere añadir un nuevo contacto especificando una lista de mascotas, primero deberá añadirse o buscar las mascotas correspondientes (y la dirección, por supuesto), y luego guardar el nuevo contacto con todos los identificadores necesarios.

- Entonces, primero se añade o busca la dirección y/o mascotas correspondientes, y se obtienen los identificadores asociados.

```
let address1 = new Address({
    …
});
address1.save().then(…

let pet1 = new Pet({
    name: "Otto",
    type: "dog"
});
pet1.save().then(…
…
```

- Después, se añade el nuevo contacto con todos los identificadores necesarios para los campos correspondientes. En cuanto al campo de mascotas, deberá ponerse los *ids* como un *array* (entre corchetes):

```
let contact1 = new Contact({
    name: "Javier",
    telephone: "966112233",
    age: 44,
    address: '5acd3c051d694d04fa26dd8b',
    pets: ['5acd3c051d694d04fa26dd90', '5acd3c051d694d04fa26dd91']
});
contact1.save().then(result => { …
```

1.1.2. Sobre la integridad referencial

La integridad referencial es un concepto asociado a las bases de datos relacionales, a través del cual los valores de una clave externa deben existir en la tabla original a la que hacen referencia. Si se aplica este concepto a una base de datos de Mongo, se podría pensar que los identificadores que se establecen en un campo que “apunta a” otra colección deben existir en esa colección… pero esto no tiene por qué ser necesariamente cierto.

En otras palabras, si se intenta añadir un nuevo contacto con un *id* de dirección que no existe en la colección de direcciones, se podrá hacer, siempre que sea un *id* válido (es decir, 12 bytes de longitud). Por lo tanto, es responsabilidad del programador asegurarse de que los identificadores que se están administrando existan en las colecciones correspondientes. Hay algunas bibliotecas auxiliares (como esta) que pueden ayudar con este problema, pero no se usarán en esta unidad.

Con respecto a la operación de eliminación, puedes encontrarte con una situación similar: si eliminas un documento al que se hace referencia de otra colección, debe cambiarse manualmente la referencia en los objetos afectados, o prohibir esta operación hasta que se hayan cambiado todas las referencias

1.2. Subdocumentos

Mongoose permite definir subdocumentos. Un subdocumento es un objeto complejo que está incrustado en otro. Por ejemplo, en lugar de definir un nuevo modelo para las mascotas o direcciones de los contactos, se podría simplemente añadir dentro de cada contacto. Sería necesario definir la mascota y el esquema de direcciones como se hizo antes:

Pero, en lugar de asignar este esquema a un modelo, simplemente se usará al definir el esquema para el contacto.

```
let addressSchema = new mongoose.Schema({
    … // Same as before
});

let petSchema = new mongoose.Schema({
    … // Same as before
});

let contactSchema = new mongoose.Schema({
    name: {
        …
    },
    telephone: {
        …
    },
    age: {
        …
    },
    address: addressSchema,
    pets: [petSchema]
});
let Contact = mongoose.model('contacts', contactSchema);
```

Presta atención a las líneas en negrita: esta es la forma de asociar un esquema como un tipo de datos de otro esquema, de modo que se definen subdocumentos. Observa también que solo se necesita un modelo para los contactos (las direcciones y las mascotas no tendrían ningún modelo ni colección asociada en la base de datos).

Por tanto, un subdocumento es un objeto incrustado en otro objeto, que es diferente a cualquier otro objeto que pueda existir (incluso si tienen los mismos campos). Con una relación simple o múltiple se puede compartir un documento de una colección entre muchos documentos de otra colección (por ejemplo, dos o más contactos que comparten la misma dirección). Pero con los subdocumentos, cada contacto tiene sus propios datos, que son diferentes de los datos del resto de contactos.

1.2.1. Insertar documentos con subdocumentos

Si se desea insertar un nuevo contacto con su dirección y mascotas, se podrá crear todo el objeto a la vez y guardarlo.

```
let contact1 = new Contact({
    name: 'Javier',
    telephone: '966112233',
    age: 44,
    address: {
```

```
        street: 'C/Pelayo',
        number: 12,
        city: 'San Vicente',
        postalCode: 03690
    },
});

contact1.pets.push({name:'Otto', type:'dog'});
contact1.pets.push({name:'Gandalf', type:'other'});
contact1.save().then(...
```

En este ejemplo se pude ver que es posible añadir subdocumentos de dos maneras: definidiendo el objeto principal (así es como se ha definido la dirección), o una vez definido el objeto principal (así es como se ha añadido la lista de mascotas).

En la base de datos se verá solo una colección, y dentro de cada documento se podrán ver sus subdocumentos incrustados.

1.3. ¿Cuándo usar relaciones o subdocumentos?

La respuesta a esta pregunta puede parecer compleja u obvia, dependiendo de lo bien que se hayan entendido los conceptos explicados hasta ahora. Observa algunas reglas simples a seguir:

- Se usarán relaciones (simples o múltiples) entre colecciones siempre que se necesite compartir el mismo documento de una colección entre varios documentos de otra colección.
- Se utilizarán subdocumentos cuando no importe compartir información o cuando la simplicidad del código sea más importante. Si se usan subdocumentos, es más fácil acceder a la información de estos subdocumentos desde el documento principal (no se necesita unir documentos o completar colecciones). Sin embargo, se puede duplicar la información en este caso (por ejemplo, dos personas que tienen la misma mascota… se necesitará crear dos objetos idénticos para ellos).

Ejercicio 1

Vuelve al proyecto *books*. Define un esquema para almacenar comentarios sobre un libro. Cada comentario constará de una fecha (tipo *Date*), el *nick* de la persona que hace el comentario y el comentario en sí (ambos cadenas). Todos los campos son obligatorios y, con respecto a la fecha, se puede establecer como valor predeterminado la fecha actual (*Date.now*).

Esta vez, no se definirá un modelo para este esquema, sino un subdocumento dentro del esquema del libro, que almacenará una serie de comentarios. Después, intenta añadir un par de libros con algunos comentarios sobre ellos.

2. Consultas avanzadas

Observa algunos conceptos avanzados sobre las consultas que se pueden crear para obtener datos de las bases de datos de Mongo.

2.1. Más sobre *populate*

Ya se ha visto el método *populate* que permite cargar la información de un documento cuando se está obteniendo otro relacionado con el primero.

```
Contact.find().populate('address').then(result => { …
```

Si tienes más campos vinculados a otros documentos, también podrás unirlos a más llamadas de *populate* para completarlos todos:

```
Contact.find().populate('address').populate('pets').then( …
```

También se puede mapear solo una parte de la información del documento poblado. Por ejemplo, si solo se necesita el nombre de las mascotas, se puede hacer esto:

```
Contact.find().populate('pets', 'name') …
```

Desde las últimas versiones de Mongoose, también se puede hacer poblaciones anidadas. Por ejemplo, supón que los contactos pueden tener una variedad de contactos como amigos. De esta manera, se puede completar la lista de amigos de un contacto, pero también los amigos de estos amigos, etc. Se puede aprender más sobre las poblaciones aquí.

2.2. Más sobre consultas

Se pueden utilizar métodos como *find*, *findOne* o *findById* para obtener documentos de una colección determinada, pero hay algunos conceptos avanzados sobre estos métodos que aún no se han tratado.

2.2.1. Otras opciones de búsqueda

Cuando se utiliza *find* o cualquier otro método similar, existen algunas opciones adicionales que permiten especificar los campos que se quieren obtener, o algún criterio de orden, o el límite máximo de resultados a obtener…

- Esta consulta obtiene todos los contactos cuyas edades se encuentran entre los 18 y los 40 años (inclusive)

```
Contact.find({age: {$gte: 18, $lte: 40}}).then(result => {
    console.log(result);
}).catch(error => {
    console.log('ERROR:', error);
});
```

Aquí puedes encontrar una lista detallada de operadores que se pueden utilizar en este tipo de consultas.

- Esta otra consulta se usa una expresión regular para hacer coincidir todos los contactos cuyo nombre contiene el texto “Juan”:

```
Contact.find({name: /Juan/}).then(result => { …
```

También se puede utilizar cualquier otro carácter especial de una expresión regular en el proceso de comparación. Por ejemplo, estas otras consultas solo obtienen los contactos cuyo nombre comienza con “Juan”:

```
Contact.find({name: /^Juan/}).then(result => { …
```

- Es posible combinar algunas condiciones con y/o operadores. Esta consulta obtiene los contactos cuyo nombre comienza con “Juan” y tienen una edad entre 18 y 40:

```
Contact.find({$and: [name: /^Juan/, age: {$gte: 18, $lte: 40}]}).then(result => { …
```

- Se pueden especificar los campos que se desean obtener. Esta consulta obtiene el nombre y la edad de los contactos cuya edad es superior a 30:

```
Contact.find({age: {$gt: 30}}, 'name age').then( …
```

Además, se puede unir la llamada de búsqueda con una llamada de selección que mapea estos campos:

```
Contact.find({age: {$gt: 30}}).select('name age').then( …
```

- Si se quiere ordenar la lista por un campo dado, se utilizará el método *sort* y se especificará el campo por el cual se ordenará el resultado y el orden (1 para ascendente, -1 para descendente). Esta expresión ordena el contacto por edad en orden descendente:

```
Contact.find().sort({age: -1}).then( …
```

Esta otra expresión también es equivalente:

```
Contact.find().sort('-age').then( …
```

- Si quieres limitar la cantidad de resultados obtenidos de una consulta, puedes usar el método *limit*, especificando cuántos resultados quieres obtener.

```
Contact.find().sort('-age').limit(5).then(...
```

Ejercicio 2

Añade una consulta al proyecto *books* que liste el título y el precio (junto con la identificación) de los 3 libros más baratos, ordenados en orden ascendente (por precio).

2.2.2. Consultas con múltiples colecciones

Las bases de datos noSQL no son adecuadas para obtener información de múltiples colecciones (si planeas hacer esto, deberías usar subdocumentos en su lugar para almacenar toda la información relevante para un documento dado).

Supón que tienes una base de datos relacional para los contactos, y quieres obtener las direcciones de aquellos cuya edad es mayor de 30 años. Podrías definir una consulta como esta:

```
SELECT * FROM addresses WHERE id IN (SELECT address FROM contacts WHERE age > 30);
```

Sin embargo, esto no es posible en MongoDB o, al menos, no tan inmediatamente. Es necesario dividir la consulta en dos partes: se necesita obtener la identificación de la dirección para las personas cuya edad es mayor de 30, y luego hacer otra consulta para obtener la información de estas direcciones:

```
Contact.find({age: {$gt: 30}}).then(resultContacts => {
    let idsAddresses = resultContacts.map(contact => contact.address);

    Address.find({_id: {$in: idsAddresses}}).then(finalResult => {
        console.log(finalResult);
    });
});
```