Tema 2. Acceso a datos con Node.js

Opciones avanzadas de MongoDB y Mongoose

Relaciones entre colecciones

Definir una relación simple (uno a uno)

- *ProyectosNode/Pruebas/PruebaMongo*
- Añadir, para cada contacto, cúal es su restaurante favorito, de forma que varios contactos pueden tener el mismo restaurante.
- Del restaurante nos interesa: *nombre*, *dirección* y *teléfono*.

Relaciones entre colecciones

```
let restauranteSchema = new mongoose.Schema({
    nombre: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    direccion: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    telefono: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        match: /^\d{9}$/
    }
});
let Restaurante = mongoose.model('restaurante', restauranteSchema);
```

Relaciones entre colecciones

- Asociamos el modelo al esquema de contactos con un nuevo campo: **restauranteFavorito**
- El tipo de dato de restauranteFavorito es **ObjectId**, lo que indica que hace referencia a un id de un documento de ésta u otra colección
- La propiedad **ref** indica a qué modelo o colección hace referencia dicho id

```
let contactoSchema = new mongoose.Schema({
    nombre: {
        ...
    },
    telefono: {
        ...
    },
    edad: {
        ...
    },
    restauranteFavorito: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'restaurante'
    }
});
let Contacto = mongoose.model('contacto', contactoSchema);
```

Relaciones entre colecciones

Definir una relación múltiple (uno a muchos)

- Relación que permite asociar a un elemento de una colección múltiples elementos de otra (o de esa misma colección).
- Por ejemplo, cada contacto puede tener conjunto de mascotas.
  - Definimos un nuevo esquema para las mascotas, que almacene su nombre y tipo (perro, gato, etc.).

Relaciones entre colecciones

```
let mascotaSchema = new mongoose.Schema({
    nombre: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    tipo: {
        type: String,
        required: true,
        enum: ['perro', 'gato', 'otros']
    }
});
let Mascota = mongoose.model('mascota', mascotaSchema);
```

Relaciones entre colecciones

- Se añade un campo nuevo en el esquema de contactos que será un array de ids, asociados al modelo de mascotas definido previamente
- Este campo nuevo es un array

```
let contactoSchema = new mongoose.Schema({
    nombre: {
        ...
    },
    telefono: {
        ...
    },
    edad: {
        ...
    },
    restauranteFavorito: {
        ...
    },
    mascotas: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: 'mascota'
    }]
});
let Contacto = mongoose.model('contacto', contactoSchema);
```

Inserciones de elementos relacionados

- Para insertar un nuevo contacto y, a la vez, especificar su restaurante favorito y/o sus mascotas:

  (1) Se añade el restaurante favorito a la colección de restaurantes, y/o las mascotas a la colección de mascotas (salvo que exista previamente, en cuyo caso obtendríamos su id)

Inserciones de elementos relacionados

```
let restaurante1 = new Restaurante({
    nombre: "La Tagliatella",
    direccion: "C.C. San Vicente s/n",
    telefono: "965678912"
});
restaurante1.save().then(...

let mascota1 = new Mascota({
    nombre: "Otto",
    tipo: "perro"
});
mascota1.save().then(...
```

Inserciones de elementos relacionados

(2) Se añade el nuevo contacto especificando el id de su restaurante favorito,añadido previamente, y/o los ids de sus mascotas en un array.

```
let contacto1 = new Contacto({
    nombre: "Nacho",
    telefono: 677889900,
    edad: 40,
    restauranteFavorito: '5acd3c051d694d04fa26dd8b',
    mascotas: ['5acd3c051d694d04fa26dd90',
               '5acd3c051d694d04fa26dd91']
});
contacto1.save().then(...
```

Sobre la integridad referencial

- La integridad referencial, en BD relacionales, garantiza que los valores de una clave ajena siempre van a existir en la tabla a la que hace referencia
- En MongoDB no es así. Los ids de los campos vinculados a otra colección no tienen por qué existir en dicha colección.
  - Es el programador es el que debe asegurarse de que los id empleados en inserciones que impliquen una referencia a otra colección existan realmente.
  - Existen algunas librerías en el repositorio NPM que podemos emplear, como por ejemplo: mongoose-id-validator

Sobre la integridad referencial

- En el caso del borrado, si queremos borrar un restaurante , debemos tener cuidado con los contactos que lo tienen asignado como restaurante favorito, ya que el id dejará de existir en la colección de restaurantes.
- En este caso, tenemos 2 opciones, aunque las dos requieren un tratamiento manual por parte del programador:
  - Denegar la operación si existen contactos con el restaurante seleccionado
  - Reasignar (o poner a nulo) el restaurante favorito de esos contactos, antes de eliminar el restaurante seleccionado

Subdocumentos

- Mongoose ofrece la posibilidad de definir **subdocumentos**.
  - Copia y pega el archivo index.js de "**PruebaMongo**", y renombralo a **index_subdocumentos.js** en esa misma carpeta.
  - Conecta con una nueva BD, que llamaremos contactos_subdocumentos, para no interferir con la BD anterior: `mongoose.connect('mongodb://localhost:27017/contactos_subdocumentos');`
  - Y reagrupa los tres esquemas (restaurantes,mascotas y contactos) para unirlos en el de contactos.

- En las propiedades restauranteFavorito y mascotas se asocia un esquema entero como tipo de dato de un campo de otro esquema, creando así subdocumentos dentro del documento principal.

- No se definen modelos ni para los restaurantes ni para las mascotas, ya que ahora no van a tener una colección propia.

- La principal diferencia entre un subdocumento y una relación entre documentos de colecciones diferentes es que el subdocumento queda embebido dentro del documento principal, y es diferente a cualquier otro objeto que pueda haber en otro documento, aunque sus campos sean iguales: se crea el restaurante para cada contacto, diferenciándolo de los otros restaurantes, aunque sean iguales

```
// Restaurantes
let restauranteSchema = new mongoose.Schema({
    ...
});

// Mascotas
let mascotaSchema = new mongoose.Schema({
    ...
});

// Contactos
let contactoSchema = new mongoose.Schema({
    nombre: {
        ...
    },
    telefono: {
        ...
    },
    edad: {
        ...
    },
    restauranteFavorito: restauranteSchema,
    mascotas: [mascotaSchema]
});
let Contacto = mongoose.model('contacto', contactoSchema);
```

Inserción de documentos son subdocumentos

Para crear y guardar un contacto que contiene como subdocumentos el restaurante favorito y sus mascotas, hay que crear todo el objeto completo, y hacer un único guardado (save).

Se muestran dos formas de rellenar los subdocumentos del documento principal:

- Sobre la marcha cuando creamos dicho documento (caso del restaurante)
- A posteriori, accediendo a los campos y dándoles valor (caso de las mascotas).

```
let contacto1 = new Contacto({
    nombre: 'Nacho',
    telefono: 966112233,
    edad: 39,
    restauranteFavorito: {
        nombre: 'La Tagliatella',
        direccion: 'C.C. San Vicente s/n',
        telefono: 961234567
    }
});
contacto1.mascotas.push({nombre:'Otto', tipo:'perro'});
contacto1.mascotas.push({nombre:'Piolín', tipo:'otros'});
contacto1.save().then(...
```

¿Cuando definir relaciones o subdocumentos?

- Emplearemos **relaciones** entre colecciones cuando queramos poder compartir un mismo documento de una colección por varios documentos de otra (por ejemplo, el caso de los restaurantes)
- Emplearemos **subdocumentos** cuando no importe dicha compartición de información, o cuando prime la simplicidad de definición de un objeto frente a la asociatividad entre colecciones (por ejemplo, si preferimos poder acceder de forma sencilla a las mascotas de un contacto, sin importar si otro contacto tiene las mismas mascotas)
  - Se produce duplicidad de información, a cambio el acceso a la misma es más sencillo.

Consultas avanzadas

Poblaciones (populate)

- Si para obtener toda la información de nuestros contactos, relacionados con las colecciones de restaurantes y mascotas hacemos esto:

```
Contacto.find().then(resultado => {
    console.log(resultado);
});
```

- Obtendremos el id de los restaurantes favoritos y de las mascotas, pero no los datos completos de las mismas.
- Para eso, hay que utilizar el método **populate** de Mongoose. Este método permite incorporar la información asociada al modelo que se le indique.

Consultas avanzadas

- Por ejemplo, si queremos incorporar al listado anterior toda la información del restaurante favorito de cada contacto, haremos algo así:

```
Contacto.find().populate('restauranteFavorito').then(resultado => {
    console.log(resultado);
});
```

- Si tuviéramos más campos relacionados, podríamos enlazar varias sentencias populate, una tras otra, para poblarlos. Por ejemplo, así poblaríamos tanto el restaurante como las mascotas:

```
Contacto.find()
.populate('restauranteFavorito')
.populate('mascotas')
.then(resultado => {
    console.log(resultado);
});
```

Consultas avanzadas

- También podemos poblar solo una parte de la información, utilizando una serie de parámetros adicionales en el método populate.
  - Por ejemplo, si queremos obtener unicamente el nombre del restaurante:

```
Contacto.find()
.populate('restauranteFavorito', 'nombre')
...
```

Consultas avanzadas

Otras opciones en las consultas

- El método find (o similares), admite opciones adicionales que permiten especificar los campos queremos obtener, los criterios de ordenación a aplicar, un límite máximo de resultados a obtener, etc.
  - Otra opción consiste en enlazar la llamada normal a find con el método select, donde se especifican los campos a obtener.

```
Contacto.find({edad: {$gt: 30}}, 'nombre edad').then(...
Contacto.find({edad: {$gt: 30}}).select('nombre edad').then(...
```

Estas 2 sentencias hacen lo mismo: mostrar el nombre y la edad de los contactos mayores de 30 años.

Consultas avanzadas

- **Ordenar**: enlazar la llamada a find con otra a sort, indicando el campo por el que ordenar y el orden (ascendente: 1, descendente: -1)

```
Contacto.find().sort({edad: -1}).then(...
Contacto.find().sort('-edad').then(...
```

- **Limitar el número de resultados**: emplear el método limit, indicando cuántos documentos obtener.

```
Contacto.find().sort('-edad').limit(5).then(...
```

Consultas avanzadas

Consultas que relacionan varias colecciones

- Obtener los datos de los restaurantes favoritos de aquellos contactos que sean mayores de 30 años.

BD SQL

```
SELECT * FROM restaurantes
WHERE id IN
(SELECT restauranteFavorito FROM contactos
WHERE edad > 30)
```

BD Mongo

```
Contacto.find({edad: {$gt: 30}}).then(resultadoContactos => {
    let idsRestaurantes =
       resultadoContactos.map(contacto => contacto.restauranteFavorito);
    Restaurante.find({_id: {$in: idsRestaurantes}})
    .then(resultadoFinal => {
        console.log(resultadoFinal);
    });
});
```

Consultas avanzadas

- MongoDB esta pensado más para utilizar subdocumentos que para relacionar colecciones. La consulta anterior utilizando subdocumentos quedaría así:

```
Contacto.find({edad: {$gt:30}}, 'restauranteFavorito')
.then(resultado => {
    console.log(resultado);
});
```