



30

Node S3 | Mongoose

Después de esta lección podrás:

1. Crear tu propia base de datos noSQL.
2. Modificar la información de tu base de datos y guardar tu información.

¿Para qué usaremos una base de datos?

Hasta el momento hemos utilizado datos guardados en archivos o en variables dentro de nuestro servidor, pero te habrás dado cuenta de que estos datos no son estables y podemos tener problemas si reiniciamos nuestro servidor.

Para conseguir consistencia en la información que tengamos en nuestro servidor, utilizaremos una **base de datos**.

MongoDB

La base de datos que usaremos en este módulo será **MongoDB** debido a que utilizarla desde Node.js y el entorno JavaScript se convierte en una tarea muy sencilla debido al sistema de guardado de documentos. MongoDB es una base de datos **no relacional**, por lo que no es

necesario definir tablas de datos ni guardaremos la información en forma de columnas y filas, sino en forma de documentos.

¿Cómo utilizaremos MongoDB desde Node.js?

Si instalamos el driver de MongoDB en nuestro proyecto de Node.js podemos conectarnos con la base de datos directamente utilizando las funciones que aprendimos en el módulo de MongoDB. Aunque esta pueda parecer una buena solución, utilizaremos una alternativa muy parecida al sistema original en forma de **librería npm** llamada **Mongoose**.

- Aquí la web de Mongoose: <https://mongoosejs.com/>

¿Qué ventajas dará Mongoose?

Como ya sabemos, MongoDB permite crear colecciones y documentos sin problema alguno, cambiando el formato de la información cuando queramos. Esto puede ser una espada de doble filo si no lo usamos correctamente, y para evitarnos problemas utilizaremos este **ODM (Object Document Mapper)** de más alto nivel.

Con Mongoose crearemos **esquemas** que definirán las **colecciones** que guardaremos en la base de datos y el **formato de los documentos** que podemos guardar en ésta. Crearemos un proyecto inicial con Express como en las sesiones anteriores e instalaremos mongoose con el comando npm correspondiente (como dependencia de producción):

```
$ npm install mongoose
```

Una vez instalado, crearemos un pequeño archivo **db.js** donde nos conectaremos a MongoDB mediante la función **connect** de mongoose:

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

// URL local de nuestra base de datos en mongoose y su nombre upgrade_class_3
const DB_URL = 'mongodb://localhost:27017/upgrade_class_3';

// Función que conecta nuestro servidor a la base de datos de MongoDB mediante mongoose
mongoose.connect(DB_URL, {
```

```
useNewUrlParser: true,  
useUnifiedTopology: true,  
});
```

Si ahora requerimos este archivo en nuestro `index.js` de esta forma:

```
const express = require('express');  
  
// Requerimos el archivo de configuración de nuestra DB  
require('./db.js');  
  
const PORT = 3000;  
const server = express();  
  
server.listen(PORT, () => {  
  console.log(`Server running in http://localhost:${PORT}`);  
});
```

Al lanzar nuestro servidor, **nos conectaremos a la base de datos** si hemos instalado y abierto el servicio de MongoDB previamente en nuestro sistema.

Ahora que tenemos el servicio conectado, vamos a crear un esquema y a empezar a llenar nuestras colecciones.

Creando nuestro primer esquema

Comenzaremos creando una carpeta `models` en nuestro proyecto en la que a partir de ahora crearemos un archivo por esquema. Por ejemplo, si queremos crear un esquema de mascotas, crearemos un archivo `Pet.js` en nuestra carpeta `models`.

- ¡Recuerda! Al modelo nos referiremos siempre con mayúsculas y en singular, es decir, si tenemos un modelo de vehículos, el archivo se llamará `Vehicle.js`.

Vamos a crear un modelo para mascotas en las que nuestros documentos tendrán nombre, especie y edad:

```
// Archivo Pet.js dentro de la carpeta models
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

// Creamos el esquema de mascotas
const petSchema = new Schema(
{
  name: { type: String, required: true },
  age: { type: Number },
  species: { type: String, required: true },
},
{
  // Esta propiedad servirá para guardar las fechas de creación y actualización de los documentos
  timestamps: true,
}
);

// Creamos y exportamos el modelo Pet
const Pet = mongoose.model('Pet', petSchema);
module.exports = Pet;
```

Como puedes ver, la clase **Schema** de mongoose acepta 2 argumentos cuando se instancia, el primero es el formato del documento que guardamos en la base de datos, y el segundo las opciones, que serán **timestamps: true** en nuestro caso.

Creando nuevos endpoints POST, PUT y DELETE

Ahora que sabemos crear nuevos endpoints en Express y tenemos control sobre nuestra base de datos, vamos a continuar aprendiendo el proceso CRUD visto hasta ahora con los endpoints restantes, para terminar nuestra primera API funcional en esta sesión.

Utilizando POST para añadir nuevos documentos a nuestra DB

Al igual que los endpoints pueden tener **query y route params**, los endpoints de tipo **POST y PUT** pueden tener una propiedad llamada **body** que utilizaremos para enviar datos a estos endpoints.

Vamos a aprovechar el último proyecto sobre mascotas y a crear un endpoint para añadir nuevas mascotas.

Hasta ahora hemos utilizado las rutas directamente en el `index.js` de nuestro servidor, pero de este momento en adelante vamos a crear una carpeta llamada `routes` en la que exportaremos cada tipo de ruta en un archivo distinto, en nuestro caso, crearemos el archivo `pet.routes.js` en el que añadiremos todas las rutas relacionadas con mascotas.

```
// Archivo pet.routes.js dentro de la carpeta routes
const express = require('express');

const Pet = require('../models/Pet');

const router = express.Router();

router.get('/', async (req, res) => {
  try {
    const pets = await Pet.find();
    return res.status(200).json(pets)
  } catch (err) {
    return res.status(500).json(err);
  }
});

module.exports = router;
```

Y ahora, ¿por qué el endpoint utiliza `'/'` en vez `'/pets'` como antes? Vamos a ver el motivo y razonar su solución.

Añadiremos en el archivo `index.js` un nuevo middleware de rutas para el endpoint `'/pets'`:

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

// Importaremos las rutas de mascotas aquí
const petRoutes = require('./routes/pet.routes');

const PORT = 3000;
const server = express();

// Aquí tendremos los endpoints y rutas de las mascotas
server.use('/pets', petRoutes);

server.use( (req, res, next) => {
  const error = new Error('Route not found');
  error.status = 404;
  next(error); // Lanzamos la función next() con un error
});

server.use((err, req, res, next) => {
  return res.status(err.status || 500).json(err.message || 'Unexpected error');
});
```

```
server.listen(PORT, () => {
  console.log(`Server running in http://localhost:${PORT}`);
});
```

Por tanto, nuestro código se comporta de la siguiente forma:

- El cliente hace una request a `http://localhost:3000/pets`
- El código de index.js llega al punto `app.use('/pets', petRoutes)` ya que las rutas coinciden.
- Al entrar en `pet.routes.js` ya se ha considerado la parte `/pets` de la ruta, por lo que el archivo empieza en su nueva base `/`.
- Se busca el endpoint GET, POST, PUT o DELETE que coincida con la nueva base de ruta y se lanza la función Response o Next para continuar o terminar el proceso en ese controlador.

Ahora que sabemos como funciona la división de rutas por archivos, vamos a crear nuestro endpoint POST para crear mascotas:

```
router.post('/create', async (req, res, next) => {
  try {
    // Crearemos una instancia de mascota con los datos enviados
    const newPet = new Pet({
      name: req.body.name,
      species: req.body.species,
      age: req.body.age,
    });

    // Guardamos la mascota en la DB
    const createdPet = await newPet.save();
    return res.status(200).json(createdPet);
  } catch (err) {
    // Lanzamos la función next con el error para que gestione todo Express
    next(err);
  }
});
```

¿Cómo accedemos a este endpoint? ¿Cómo enviamos información a req.body?

Al endpoint se accederá a través de `http://localhost:3000/pets/create` pero, no podremos hacerlo a través del navegador todavía, ya que necesitaríamos un formulario o una función `fetch` que envíe datos en formato **JSON**, así que vamos a utilizar **Postman** para simular peticiones.

Vamos a añadir una nueva request POST, con el endpoint que acabamos de crear, y enviaremos los datos en formato `raw` y `JSON` de la siguiente manera:

The screenshot shows the Postman interface with a red box highlighting the 'Body' tab. Below it, the 'Content-Type' dropdown is set to 'JSON'. The JSON payload is as follows:

```
1 [ {  
2   "name": "Rufus",  
3   "age": 6,  
4   "species": "dog"  
5 } ]
```

Si tenemos todo configurado correctamente, podremos enviar la petición a través de Postman y deberíamos poder crear la nueva mascota en la base de datos, pero **¡tendremos un error!**

Esto se debe a que `req.body` no contendrá los datos que hemos enviado al servidor porque no estamos utilizando ningún middleware para "parsear" la información. Para ello, añadiremos estas líneas en el archivo `index.js` antes de añadir las rutas.

Estas son funciones propias de express que transforman la información enviada como JSON al servidor de forma que podremos obtenerla en `req.body`.

```
server.use(express.json());  
server.use(express.urlencoded({ extended: false }));
```

Repetiremos el proceso para confirmar que ahora podemos crear nuevas mascotas:

```
1 {  
2   "_id": "5eada22bb103334252874b00",  
3   "name": "Rufus",  
4   "species": "dog",  
5   "age": 6,  
6   "createdAt": "2020-05-02T16:39:07.252Z",  
7   "updatedAt": "2020-05-02T16:39:07.252Z",  
8   "__v": 0  
9 }
```

Utilizando PUT para editar un documento de nuestra DB

Como ya sabemos crear endpoints de tipo POST en nuestro servidor, vamos a editar una mascota dada su **id** utilizando los métodos PUT.

Para ello, crearemos un nuevo endpoint `/pets/edit` en que enviaremos la id nueva y los campos que queremos modificar en el documento de la colección Pet.

```
router.put('/edit', async (req, res, next) => {  
  try {  
    const id = req.body.id;  
  
    const updatedPet = await Pet.findByIdAndUpdate(  
      id, // La id para encontrar el documento a actualizar  
      { name: req.body.name }, // Campos que actualizaremos  
      { new: true } // Usando esta opción, conseguiremos el documento actualizado cuando se complete el update  
    );  
  
    return res.status(200).json(updatedPet);  
  } catch (err) {  
    next(err);  
  }  
});
```

Cambiaremos el nombre del perro que acabamos de crear de Rufus a Maximus:

The screenshot shows the Postman interface. The method is set to PUT, and the URL is http://localhost:3000/pets/edit. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   "id": "Seada22bb103334252874b00",  
3   "name": "Maximus"  
4 }
```

Si enviamos la request de tipo PUT y hemos añadido todo correctamente a nuestro servidor, nuestro perro habrá cambiado de nombre 🐶

Utilizando DELETE para eliminar un documento de la DB

Aunque podamos encontrarlo en algún proyecto, no es correcto enviar body a nuestro endpoint de tipo DELETE, por lo tanto utilizaremos un parámetro de ruta **/:id** para identificar el elemento que queremos eliminar de nuestra DB.

```
router.delete('/:id', async (req, res, next) => {  
  try {  
    const id = req.params.id;  
  
    // No será necesaria asignar el resultado a una variable ya que vamos a eliminarlo  
    await Pet.findByIdAndDelete(id);  
    return res.status(200).json('Pet deleted!');  
  } catch (err) {  
    next(err);  
  }  
});
```

Con esto, podremos lanzar la petición al endpoint usando el método DELETE y podremos ver como hemos eliminado el elemento de nuestra base de datos.

The screenshot shows the Postman interface with a successful DELETE request. The URL is `http://localhost:3000/pets/5eada22bb103334252874b00`. The 'Params' tab is selected, showing a single query parameter 'Key' with value 'Value'. The 'Body' tab is selected, containing the JSON response: `1 "Pet deleted!"`.

¡Hemos aprendido a crear una API CRUD completa! 🎉

Aprendiendo a relacionar modelos entre si 💡

Como último paso en la clase de hoy, relacionaremos varios modelos entre si utilizando un nuevo tipo de variable de mongoose, ya que hasta este momento hemos visto como crear campos de tipo String y Number.

Vamos a suponer que tenemos casas de acogida y las mascotas, en este caso tendremos dos modelos, Shelter y Pet. La relación de estos modelos será tal que una casa de acogida tenga un array de ids representando a cada animal que se encuentra en ella. Veamos el nuevo modelo

`Shelter.js` :

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

const Schema = mongoose.Schema;

const shelterSchema = new Schema(
{
  name: { type: String, required: true },
  location: { type: String, required: true },
  // Tipo mongoose Id y referencia al modelo Pet
  pets: [{ type: mongoose.Types.ObjectId, ref: 'Pet' }],
})
```

```

    },
    {
      timestamps: true,
    }
);

const Shelter = mongoose.model('Shelter', shelterSchema);
module.exports = Shelter;

```

¡Ya tenemos el modelo Shelter para la casa de acogida! 🏠

Vamos a crear un endpoint POST para las casas de acogida en el que crearemos una nueva casa, donde crearemos una nueva casa con estos datos:

```
{
  name: "Happy dogs",
  location: "Madrid",
  pets: [],
}
```

¿Cómo añadimos una nueva mascota?

Crearemos un endpoint PUT que reciba la id del Shelter y la id de la mascota que queremos añadir, de la siguiente forma:

```

const express = require('express');

const Shelter = require('../models/Shelter');

const router = express.Router();

router.post('/create', async (req, res, next) => {
  try {
    const newshelter = new Shelter({
      name: req.body.name,
      location: req.body.location,
    });

    const createdShelter = await newshelter.save();
    return res.status(200).json(createdShelter);
  } catch (err) {
    next(err);
  }
});

```

```

router.put('/add-pet', async (req, res, next) => {
  try {
    const shelterId = req.body.shelterId;
    const petId = req.body.petId;

    const updatedShelter = await Shelter.findByIdAndUpdate(
      shelterId,
      { $push: { pets: petId } },
      { new: true }
    );

    return res.status(200).json(updatedShelter);
  } catch (err) {
    next(err);
  }
});

module.exports = router;

```

Como puedes ver, si tomamos el atributo **\$push** como referencia, podemos añadir el nombre del campo al que hacer push dentro de un array de valores. En este caso, **petId** será añadido a **pets**.

The screenshot shows a POST request in Postman to `http://localhost:3000/shelter/add-pet`. The **Body** tab is selected, showing a raw JSON payload:

```

1 {
2   "shelterId": "Seadaa890f335d7b0fae67b2",
3   "petId": "5ead7edc734ae6cfbf1aaaae3"
4 }

```

The response body shows the updated shelter document:

```

1 {
2   "pets": [
3     "5ead7edc734ae6cfbf1aaaae3"
4   ],
5   "_id": "Seadaa890f335d7b0fae67b2",
6   "name": "Happy dogs",
7   "location": "Madrid",
8   "createdAt": "2020-05-02T17:14:49.209Z",
9   "updatedAt": "2020-05-02T17:15:30.876Z",
10  "__v": 0
11 }

```

Ahora, si creamos un endpoint GET para el Shelter, obtendremos todos los documentos de la colección, y estos tendrán un array de mascotas con ids. Podemos llenar el array con las mascotas correspondientes utilizando la función populate:

```
router.get('/', async (req, res, next) => {
  try {
    const shelters = await Shelter.find().populate('pets');

    return res.status(200).json(shelters);
  } catch (err) {
    next(err);
  }
});
```

Ahora, cada vez que recuperemos los Shelters del servidor, estarán los datos de las mascotas en este array:

```
[
  {
    "pets": [
      {
        "_id": "5ead7edc734ae6cfbf1aaaae3",
        "name": "Curro",
        "age": 3,
        "species": "dog",
        "__v": 0,
        "createdAt": "2020-05-02T14:08:28.887Z",
        "updatedAt": "2020-05-02T14:08:28.887Z"
      }
    ],
    "_id": "5eadaa890f335d7b0fae67b2",
    "name": "Happy dogs",
    "location": "Madrid",
    "createdAt": "2020-05-02T17:14:49.209Z",
    "updatedAt": "2020-05-02T17:15:30.876Z",
    "__v": 0
  }
]
```

Creando endpoints para recuperar información de nuestra base de datos

Si volvemos a lo que estábamos haciendo en la clase anterior, recordarás que aprendimos a crear endpoints GET con los que recoger información del servidor. Vamos a aprender a usar mongoose

para encontrar información de los siguientes modos:

- Todos los elementos de una colección.
- Un elemento dada su id.
- Uno o más elementos dado el valor de algún campo.

Crearemos un endpoint `/pets` en nuestro `index.js`:

```
const Pet = require('../models/Pet');

const router = express.Router();

router.get('/pets', (req, res) => {
  return Pet.find()
    .then(pets => {
      // Si encontramos las mascotas, las devolveremos al usuario
      return res.status(200).json(pets);
    })
    .catch(err => {
      // Si hay un error, enviaremos por ahora una respuesta de error.
      return res.status(500).json(err);
    });
});

server.use('/', router);
```

Si hacemos una petición al servidor en el endpoint `/pets` recuperaremos un array con todas las mascotas que hemos guardado en la base de datos.

→ **El endpoint podemos resolverlo también usando `async/await` como veremos ahora:**

```
router.get('/pets', async (req, res) => {
  try {
    const pets = await Pet.find();
    return res.status(200).json(pets)
  } catch (err) {
    return res.status(500).json(err);
  }
});
```

Una vez tenemos el primer endpoint, hagamos uno para encontrar mascotas por su id, que llamaremos `/pets/:id` y recibirá la información del parámetro `id` por medio de `req.params`:

```

router.get('/pets/:id', async (req, res) => {
  const id = req.params.id;

  try {
    const pet = await Pet.findById(id);

    if (pet) {
      return res.status(200).json(pet);
    } else {
      return res.status(404).json('No pet found by this id');
    }
  } catch (err) {
    return res.status(500).json(err);
  }
});

```

Si sacamos la id de alguna de las mascotas (**el campo _id**) y hacemos una petición a <http://localhost:3000/pets/5ead7edc734ae6cfbf1aaae3> (en el ejemplo anterior la id era esta cadena para el primer elemento), veremos como obtenemos la información de esta mascota solamente.

¿Y si ahora queremos obtener una mascota dada su especie? ¡Vamos a buscar las mascotas que sean tengan **dog** como especie 🐶:

```

router.get('/pet/species/:species', async (req, res) => {
  const species = req.params.species;

  try {
    const petsBySpecies = await Pet.find({ species: species });
    return res.status(200).json(petsBySpecies);
  } catch (err) {
    return res.status(500).json(err);
  }
});

```

Si hacemos una petición a <http://localhost:3000/pets/species/dog> obtendremos todos los animales que sean de la especie perro 🐶

Vamos a buscar ahora las mascotas por su edad, por ejemplo, las menos de 5 años:

```
router.get('/pet/age/:age', async (req, res) => {
  const age = req.params.age;

  try {
    const petsByAge = await Pet.find({ age: { $lt: age } });
    return res.status(200).json(petsByAge);
  } catch (err) {
    return res.status(500).json(err);
  }
});
```

Si hacemos una request GET a <http://localhost:3000/pets/age/5> encontraremos las mascotas con menos de 5 años debido a la utilidad que acabamos de aprender.

- Si usamos **\$lt (less than)** encontraremos valores menores al que usemos.
- Si usamos **\$lte (less than equal)** encontraremos valores menores o igual al usado.
- Si usamos **\$gt (greater than)** encontraremos los valores mayores al usado.
- Si usamos **\$gte (greater than equal)** encontraremos los valores mayores e iguales al usado.