

## Guía 3. Servicio Web CRUD tipo REST con MongoDB

Partiendo del esqueleto de **Servicio Web** (WS) implementado en la anterior guía ([WS Product](#)), vamos a crear un sencillo WS que proporcionará una **API RESTful** para atender los típicos comandos **CRUD**. Este servicio lo llamaremos [WS CRUD](#).

Gracias a este servicio se podrá *Crear, Leer, Actualizar y Eliminar* **elementos** de diferentes **colecciones** (tablas) de una base de datos (**DB**). El servicio también debe permitir *Crear* nuevas **colecciones** en la DB.

Los requerimientos del servicio [WS CRUD](#) son:

- El servicio permite gestionar una base de datos en la que se podrán crear diferentes **colecciones** o categorías de elementos (lo que llamaríamos tradicionalmente tablas).
- Por cada **colección** creada, se podrán **añadir/leer/modificar/eliminar elementos** (registros de la tabla).
- La estructura de cada elemento (al contrario de lo que ocurre en una DB estructurada) es abierta (no se definirán **modelos** para la DB).
- El formato con el que se interactuará con la DB es mediante objetos **JSON**.
- La DB con la que trabajaremos se denominará **"SD"**.

Como paso previo:

- Se tendrá que instalar el gestor de base de datos que vamos a emplear. En este caso se trata de **MongoDB**.
- Es fundamental entender desde un punto de vista técnico [qué es y cómo funciona](#) un *Servicio Web* (**WS**), por lo que se deberá investigar sobre los WS en general, sobre los WS tipo **REST**, y sobre los WS tipo **RESTful**.
- También se debe investigar acerca del funcionamiento de las bases de datos no estructuradas con **MongoDB**, fundamentalmente del funcionamiento de la biblioteca **mongojs** para npm.

Como en las guías previas, seguiremos documentando los aspectos más importantes de la práctica a través de una memoria. En este caso seguimos con la **Memoria 1**, dedicada al [WS CRUD](#).

Esta nueva actualización de la memoria la denominaremos **Memoria 1 (rev3)**, y puede titularse: **"Memoria 1 (rev3). Backend – WS CRUD"**.

En la actualización de la **memoria**, en el apartado **"Investigación"**, se recogerá un resumen de los **aspectos técnicos** investigados que resulten de utilidad para desarrollar el proyecto.

Igualmente es importante expresar formalmente el **diseño** del servicio propuesto. Para ello se deberá incorporar nuevamente en la **memoria**, en la sección **"Documentación del Servicio"**, los diagramas actualizados que cubran todas las vistas, recordemos que son: [casos de uso](#), [documentación del API](#), [arquitectura distribuida](#), [diagrama de secuencia](#), y [arquitectura física de despliegue](#).

El contenido del resto de la guía se estructura de la siguiente manera:

1. Diseño de la interfaz del Servicio Web
2. Instalación y ejecución de MongoDB
3. Implementación del servicio CRUD
4. Prueba del servicio

## Diseño de la interfaz del Servicio Web

Antes de comenzar debemos especificar cuál será la interfaz (**API**) que se va a construir para que los clientes puedan acceder al servicio (**WS**) CRUD que vamos a implementar.

El **API RESTful** que vamos a implementar se puede resumir en la siguiente **ruta o end-point**:

`/api/:coleccion/:id`

Esta simplicidad se debe a que una interfaz **RESTful** aprovecha todo el potencial de los métodos **HTTP**, evitando tener que incorporar verbos en la URL.

De esta forma, para obtener un recurso (puede ser un determinado elemento u objeto de una base de datos) se emplea el método HTTP **GET**, para crear un nuevo elemento o recurso utilizaremos **POST**, para actualizar un elemento emplearemos el método **PUT** y, finalmente, para eliminar un producto, nos basaremos en el método **DELETE**. Una forma más detallada (aunque sigue siendo resumida) de especificar la interfaz se muestra en la siguiente tabla:

Verbo HTTP	Ruta	Descripción
<b>GET</b>	<code>/api</code>	Obtenemos todas las colecciones existentes en la DB.
<b>GET</b>	<code>/api/{coleccion}</code>	Obtenemos todos los elementos de la tabla <code>{coleccion}</code> .
<b>GET</b>	<code>/api/{coleccion}/{id}</code>	Obtenemos el elemento indicado en <code>{id}</code> de la tabla <code>{coleccion}</code> .
<b>POST</b>	<code>/api/{coleccion}</code>	Creamos un nuevo elemento en la tabla <code>{coleccion}</code> .
<b>PUT</b>	<code>/api/{coleccion}/{id}</code>	Modificamos el elemento <code>{id}</code> de la tabla <code>{coleccion}</code> .
<b>DELETE</b>	<code>/api/{coleccion}/{id}</code>	Eliminamos el elemento <code>{id}</code> de la tabla <code>{coleccion}</code> .

En cualquier caso, es imprescindible documentar el API de la forma más sencilla, pero también de la forma más rigurosa posible. Para ello, tendremos que completar una documentación para cada *ruta o end-point* definidos en el API siguiendo como ejemplo la siguiente plantilla:

<b>GET,</b> <b>POST,</b> <b>PUT,</b> <b>DELETE,</b> ...	<code>/api/xxx/{paramA}/yyy?paramB=abc</code>	Breve descripción del servicio o funcionalidad que ofrece el end-point
<b>Solicitud (HTTP Request):</b>  <b>Cabeceras (Headers):</b> Content-Type: "application/json" Authorisation: bearer <<ACCESS_TOKEN (tipo JWT)>>		

**Parámetros (Parameters):**

```
paramA: xxxxxx
paramB: xxxxx
```

**Cuerpo (Body):**

```
{
  abcd: defg,
  ijklm: hijk
}
```

**Respuestas (HTTP Response):****Código de estado:**

200, 201, 400, 401, 500, ... (suele haber más de una respuesta...)

**Cabeceras (Headers):**

```
Conten-Type: "application/json"
...
```

**Cuerpo (Body):**

```
{
  result: 'OK',
  YYYY: YYYY
}
```

Recuerda que en la **memoria**, además de documentar el **API**, se deberá incorporar los esquemas que cubran todas las vistas estudiadas: [diagrama de casos de uso](#), [arquitectura del sistema](#), [diagrama de secuencia](#), y [arquitectura física de despliegue](#).

## Instalación y ejecución de MongoDB



**Nota:** dados los problemas que puede acarrear la instalación de las nuevas aplicaciones propuestas, es fundamental que en este punto se cree un [nuevo CLON de la máquina virtual](#). De esta forma, si algo va mal, siempre podremos crear nuevos clones para hacer nuevas pruebas e instalaciones hasta que logremos que todo funcione.

Comenzaremos con la instalación de nuestra base de datos no estructurada (aunque se puede trabajar con cualquier tipo de base de datos).

En esta guía suponemos que trabajamos en **Ubuntu 22.04.3**, si no es el caso, se recomienda buscar instrucciones por Internet y seguirlas con *copy+paste*. Algunos enlaces interesantes son:

<https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-ubuntu/>

<https://www.mongodb.com/download-center/community>

[https://juanda.gitbooks.io/webapps/content/api/creacion\\_de\\_una\\_api\\_con\\_nodejs.html](https://juanda.gitbooks.io/webapps/content/api/creacion_de_una_api_con_nodejs.html)

Se puede encontrar diferentes formas de instalar **MongoDB**. En nuestro caso, optamos por el siguiente método probado en la última versión de **Ubuntu 22.04.3**.

En primer lugar, instalamos las dependencias:

```
$ wget http://archive.ubuntu.com/ubuntu/pool/main/o/openssl/libssl1.1_1.1.1f-1ubuntu2_amd64.deb
$ sudo dpkg -i libssl1.1_1.1.1f-1ubuntu2_amd64.deb
```

Acto seguido, comenzamos con la instalación de **MongoDB**:

```
$ sudo apt-get install gnupg curl
$ curl -fsSL https://www.mongodb.org/static/pgp/server-7.0.asc | \
  sudo gpg -o /usr/share/keyrings/mongodb-server-7.0.gpg \
  --dearmor
$ echo "deb [ arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-server-7.0.gpg ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/7.0 multiverse"
| sudo tee /etc/apt/sources.list.d/mongodb-org-7.0.list
$ sudo apt-get update
$ sudo apt-get install -y mongodb-org
```

Tras la instalación debemos iniciar el **servidor** de base de datos (**mongod**). El inicio (**start**) y posterior gestión del servicio (**status**, **stop**, **restart**, **disable**, **enable**) lo realizaremos mediante el comando **systemctl**:

```
$ sudo systemctl start mongod
```

También podemos verificar el funcionamiento de la base de datos de la siguiente forma:

```
$ sudo systemctl status mongod
● mongod.service - MongoDB Database Server
   Loaded: loaded (/lib/systemd/system/mongod.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2024-02-07 13:10:20 CET; 12min ago
     Docs: https://docs.mongodb.org/manual
    Main PID: 2894 (mongod)
      Memory: 264.2M
         CPU: 6.688s
    CGroup: /system.slice/mongod.service
            └─2894 /usr/bin/mongod --config /etc/mongod.conf

feb 07 13:10:20 SD2024 systemd[1]: Started MongoDB Database Server.
feb 07 13:10:22 SD2024 mongod[2894]: {"t":{"$date":"2024-02-07T12:10:22.172Z"},"s":"I",  "c":"CONTROL",  "id":7484500,>
```

Finalmente, en otra terminal podemos abrir el **cliente de la base de datos** (**mongo**) y probar directamente desde la terminal comandos para gestionarla:

```
<Ctrl+Alt+T>
$ mongosh --host 127.0.0.1:27017
> show dbs
admin 0.000GB
config 0.000GB
local 0.000GB
```

**Nota:** en realidad, cuando no ponemos nada al iniciar el **cliente** **mongosh**, realmente le estamos indicando que el **servidor** **mongod** está en nuestra máquina, concretamente en el puerto 27017. La instrucción sería equivalente a:

```
$ mongosh --host 127.0.0.1:27017
```

Dentro del proyecto tendremos que instalar la biblioteca **mongodb** para trabajar desde nuestro proyecto con esta base de datos y, en nuestro caso, la biblioteca **mongojs** que nos simplificará la forma de acceder a **MongoDB** desde nuestro código **javascript** (en este caso hemos elegido **mongojs** por su simplicidad, pero es más versátil y conocida la biblioteca **mongoose** que utilizaremos en la guía de refactorización del código).

```
<Ctrl+Alt+T>
$ cd node/api-crud
$ npm i -S mongodb
$ npm i -S mongojs
```

Ahora ya estamos preparados para comenzar con la implementación del servicio.

**Nota:** no confundir el **servidor de la base de datos (mongod)** con el **cliente de la base de datos (mongosh)**.

## Implementación del Servicio CRUD

Una vez tenemos todas las herramientas y bibliotecas que necesitamos, escribiremos en el archivo **index.js** el código **JavaScript** que implementará la funcionalidad esperada de nuestro servidor.

**Nota:** debemos recordar iniciar el **servidor** de base de datos **mondoDB**, y el **servicio API REST**. También podemos iniciar en otra terminal un **cliente mongosh** para gestionar la base de datos y verificar que todos los cambios realizados a través del API REST, realmente surten efecto en la base de datos.

```
<Ctrl+Alt+T>
$ sudo systemctl start mongod

<Ctrl+Alt+T>
$ cd node/api-crud
$ npm start

<Ctrl+Alt+T>
$ mongosh --host localhost:27017
```

Abrimos nuestro editor en una nueva terminal y comenzamos a programar el servicio **WS CRUD** en el archivo **index.js**:

```
<Ctrl+Alt+T>
$ cd node/api-crud
$ code .
```

Comenzamos con la implementación de nuestro API. El primer paso consiste en importar el módulo **mongojs** instalado previamente para facilitar el acceso a la DB.

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

A partir de esta biblioteca de funciones, conectamos con la base de datos que se denominará **"SD"**.

```
var db = mongojs("SD");
```

Por supuesto, podemos incluir más parámetros en la conexión, como la IP del servidor, el usuario, etc.

```
var db = mongojs('username:password@example.com/SD');
```

Para lograr que la API pueda dar soporte a múltiples colecciones, crearemos un **middleware** cuya función de **callback** actuará cuando desde el cliente se introduzca en la ruta una **"coleccion"**. Esta función de callback, declarada implícitamente, creará una nueva propiedad en la petición (**req**) que denominaremos **collection** (por lo que accederemos a ellas a través de **req.collection**). Esta propiedad apuntará al método de la biblioteca **mongojs** que permite seleccionar la colección deseada (**db.collection**). La colección la encontramos, en este caso, en el argumento **coleccion** de la función **callback**.

```
(req, res, next, coleccion) => {
```

En segundo lugar, como se trata de un **middleware**, debemos asegurar que continúe la ejecución del código. Esto lo logramos devolviendo la función **next()**. El código completo del **middleware** es:

```
app.param("coleccion", (req, res, next, coleccion) => {
  req.collection = db.collection(coleccion);
  return next();
});
```

La estrategia general para desarrollar la funcionalidad de nuestra **API RESTful** a partir de los diferentes métodos **HTTP** y de las diferentes rutas definidas para la **API REST** es la siguiente: creamos una función de **callback** para cada **método** y **ruta**. Estas funciones serán definidas de forma implícita y tendrán como argumentos un petición (**req**), una respuesta (**res**) y una función **next()** que identifica el siguiente **middleware** a ejecutar.

```
app.método('ruta', (req, res, next) => {
  // Invocamos a la función de la biblioteca mongojs que corresponda
});
```

El esquema general de la función **callback** (conocida como **controlador**) de cada **ruta** será, a su vez, la siguiente:

```
let elementoId = req.params.id;
let elementoData = req.body;
```

Generalmente estos argumentos serán el identificador del elemento concreto con el que deseamos trabajar, y los datos de un elemento que estarían en el **body**.

A continuación, invocaremos el método de la biblioteca **mongojs** que interese en cada momento para realizar nuestra labor (**find**, **findOne**, **save**, **update**, **remove**), pasándole los argumentos concretos que precise y definiendo una nueva función de **callback** implícita que atienda el resultado de la invocación.

```
req.collection.métodoMongojs( /*argumentos en req*/, (err, result) => {
  if (err) return next(err);
  res.json(result);
});
```

La estrategia de esta nueva función de **callback** es la siguiente: si se ha producido un error, terminaremos la ejecución retransmitiendo dicho error (**err**).

```
if (err) return next(err);
```

En caso de que no haya error, devolvemos el resultado (**result**) en la respuesta (**res**) en formato **JSON**.

```
res.json(result);
```

Si lo ponemos todo junto, tendremos una estructura general, por ejemplo, para la ruta **PUT**, del tipo:

```
app.put('/api/:coleccion/:id', (req, res, next) => {
  let elementoId = req.params.id;
  let elementoData = req.body;

  req.collection.métodoDeMongojs( /*elementoID, elementoData, ...*/, (err, result) => {
    if (err) return next(err);
    res.json(result);
  });
});
```

Teniendo en cuenta que existe una correlación directa entre los *métodos HTTP* y las *funciones* que proporciona la biblioteca **mongojs** para gestionar la base de datos **mongodb** (**GET** con **find**, **POST** con **save**, **PUT** con **update**, y **DELETE** con **remove**); y teniendo también en cuenta la estrategia antes descrita para el diseño de nuestras funciones, a continuación, podemos ver cómo quedaría todo el código de nuestro **API RESTful**.

```
'use strict'

// declaraciones
const port = process.env.PORT || 3000

const express = require('express');
const logger = require('morgan');
const mongojs = require('mongojs');

const app = express();

var db = mongojs("SD"); // Enlazamos con la DB "SD"
var id = mongojs.ObjectId; // Función para convertir un id textual en un objectID

// middlewares
app.use(logger('dev')); // probar con: tiny, short, dev, common, combined
app.use(express.urlencoded({ extended: false })) // parse application/x-www-form-urlencoded
app.use(express.json()) // parse application/json

// añadimos un trigger previo a las rutas para dar soporte a múltiples colecciones
app.param("coleccion", (req, res, next, coleccion) => {
  console.log('param /api/:coleccion');
  console.log('colección: ', coleccion);

  req.collection = db.collection(coleccion);
  return next();
});

// routes
app.get('/api', (req, res, next) => {
  console.log('GET /api');
  console.log(req.collection);

  db.getCollectionNames((err, colecciones) => {
    if (err) return next(err);
    res.json(colecciones);
  });
});

app.get('/api/:coleccion', (req, res, next) => {
  req.collection.find((err, coleccion) => {
    if (err) return next(err);
    res.json(coleccion);
  });
});

app.get('/api/:coleccion/:id', (req, res, next) => {
  req.collection.findOne({ _id: id(req.params.id) }, (err, elemento) => {
    if (err) return next(err);
    res.json(elemento);
  });
});

app.post('/api/:coleccion', (req, res, next) => {
  const elemento = req.body;

  if (!elemento.nombre) {
    res.status(400).json({
      error: 'Bad data',
      description: 'Se precisa al menos un campo <nombre>'
    });
  } else {
    req.collection.save(elemento, (err, coleccionGuardada) => {
      if (err) return next(err);
      res.json(coleccionGuardada);
    });
  }
});

app.put('/api/:coleccion/:id', (req, res, next) => {
  const elementoId = req.params.id;
  const elementoNuevo = req.body;
```

```

    req.collection.update(
      { _id: id(elementoId) },
      { $set: elementoNuevo },
      { safe: true, multi: false },
      (err, elementoModif) => {
        if (err) return next(err);
        res.json(elementoModif);
      });
  });

app.delete('/api/:coleccion/:id', (req, res, next) => {
  const elementoId = req.params.id;

  req.collection.remove({ _id: id(elementoId) }, (err, resultado) => {
    if (err) return next(err);
    res.json(resultado);
  });
});

// Iniciamos la aplicación
app.listen(port, () => {
  console.log(`API REST ejecutándose en http://localhost:${port}/api/:coleccion/:id`);
});

```

## Prueba del servicio

**Nota:** se supone que ya hemos iniciado la ejecución de nuestra aplicación a través de **nodemon** para que gestione los cambios en el código fuente, así como la base de datos **mongoDB**.

Iniciamos **Postman** y vamos creando, probando y guardando las diferentes rutas del API que ofrece nuestro servidor. A continuación, veremos algunos ejemplos.

Ejemplo de creación/incorporación con **Postman** mediante una llamada **POST**

### Petición

**POST** http://localhost:3000/api/familia

**Cabecera** (raw, text): Content-Type:application/json

**Cuerpo** (raw, JSON)

```

{
  "tipo": "Hermano",
  "nombre": "Pepe",
  "edad": 46
}

```

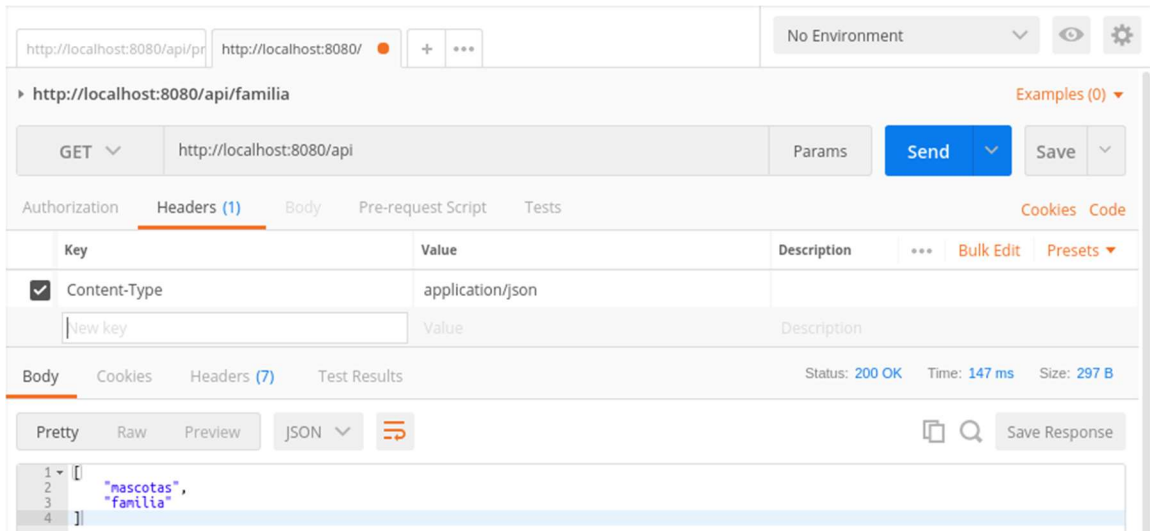
Ejemplo de solicitudes de información (tipo **GET**) tanto de colecciones existentes, como de los registros que contienen cada una de ellas. En nuestro caso obtendremos las colecciones "familia" y "mascotas". En las siguientes tres figuras podemos ver la solicitud y el resultado utilizando **Postman**:

### Petición

**GET** http://localhost:3000/api

**Captura de la respuesta**





http://localhost:8080/api/pr http://localhost:8080/ + ... No Environment

http://localhost:8080/api/familia Examples (0)

GET http://localhost:8080/api Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Cookies Code

Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Content-Type	application/json				
New key	Value	Description			

Body Cookies Headers (7) Test Results Status: 200 OK Time: 147 ms Size: 297 B

Pretty Raw Preview JSON Save Response

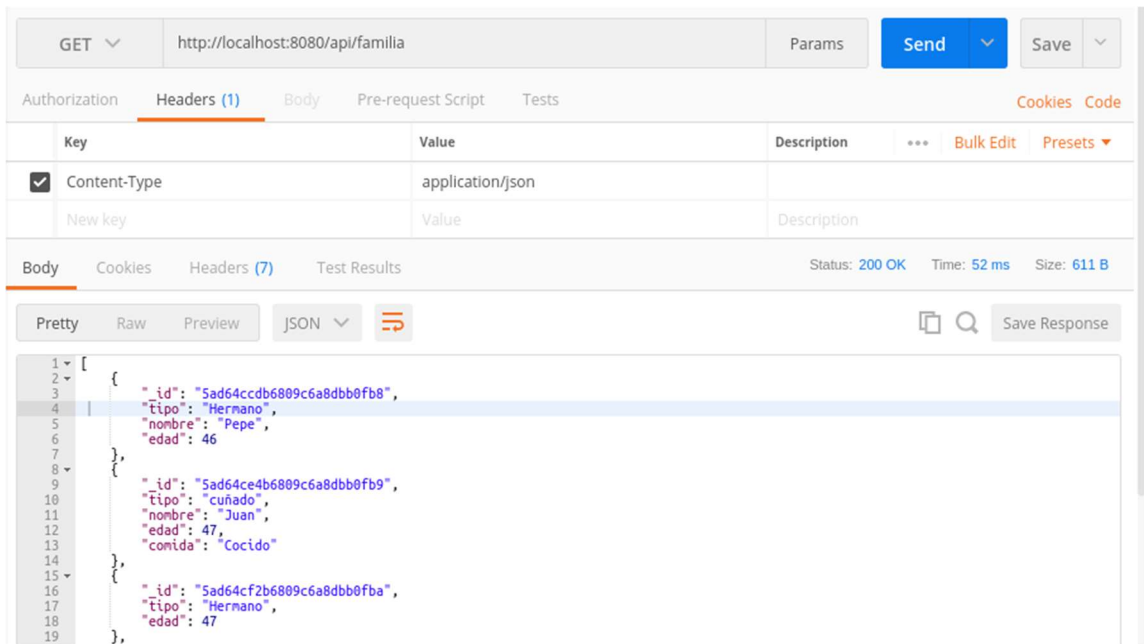
```

1  [
2    "mascotas",
3    "familia"
4  ]

```

**Petición**

GET http://localhost:3000/api/familia

**Captura de la respuesta**


GET http://localhost:8080/api/familia Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Cookies Code

Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Content-Type	application/json				
New key	Value	Description			

Body Cookies Headers (7) Test Results Status: 200 OK Time: 52 ms Size: 611 B

Pretty Raw Preview JSON Save Response

```

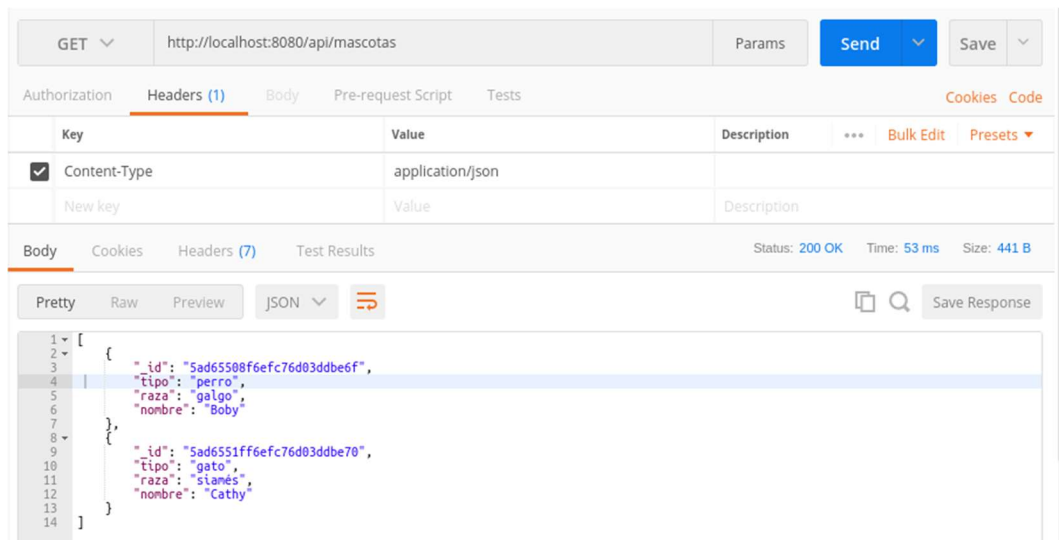
1  [
2    {
3      "_id": "5ad64ccdb6809c6a8dbb0fb8",
4      "tipo": "Hernando",
5      "nombre": "Pepe",
6      "edad": 46
7    },
8    {
9      "_id": "5ad64ce4b6809c6a8dbb0fb9",
10     "tipo": "cuñado",
11     "nombre": "Juan",
12     "edad": 47,
13     "comida": "Cocido"
14   },
15   {
16     "_id": "5ad64cf2b6809c6a8dbb0fba",
17     "tipo": "Hernando",
18     "edad": 47
19   }
20 ]

```

**Petición**

GET http://localhost:3000/api/mascotas

**Captura de la respuesta**

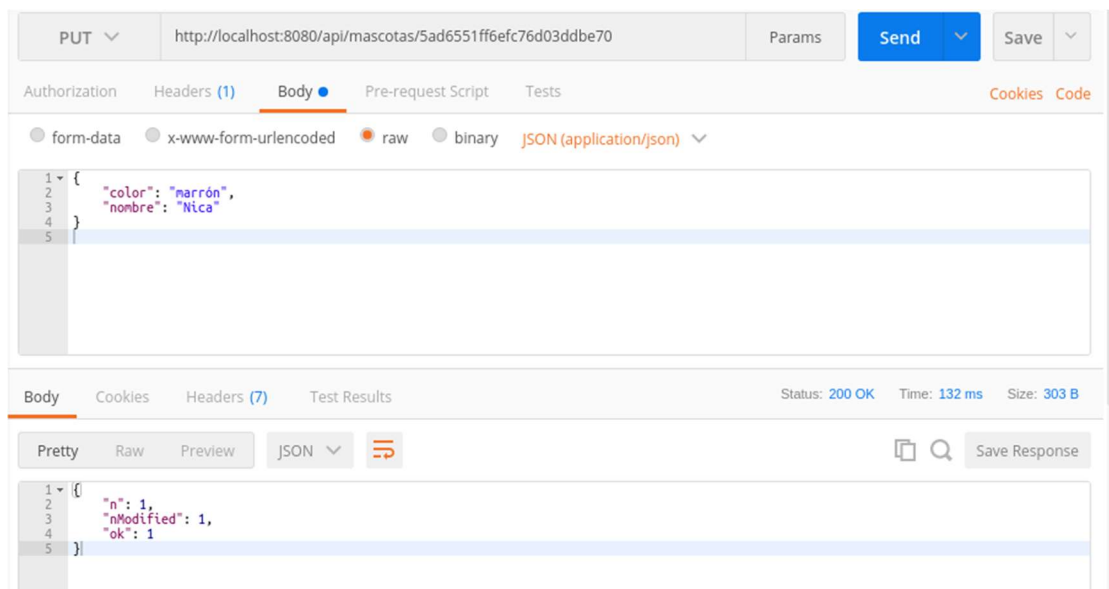


Ejemplo de actualización (método **PUT**) de los datos de un registro (modificamos un campo y creamos un campo nuevo). Posteriormente volvemos a consultar (método **GET**) la colección para comprobar, tanto de colecciones existentes, como los registros que contienen cada una de ellas.

#### Petición

**PUT** `http://localhost:3000/api/mascotas/5ad6551ff6efc76d03ddbe70`

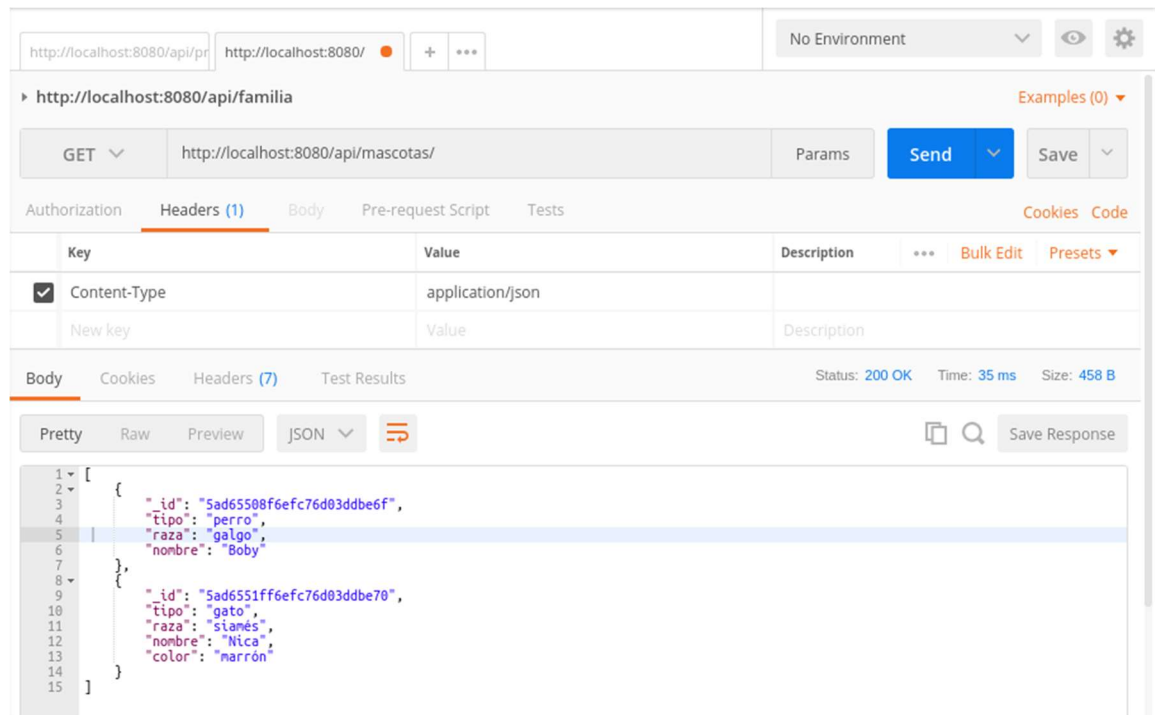
#### Captura de la respuesta



#### Petición

**GET** `http://localhost:3000/api/mascotas`

#### Captura de la respuesta



A continuación, veremos cómo utilizar el cliente de **mongoDB** (llamado **mongo**) para verificar que toda la información que hemos creado se encuentra realmente en la base de datos. Para ello podemos seguir el siguiente ejemplo mediante el cliente **mongosh**, desde una terminal de texto.

```
<Ctrl+Alt+T>
$ mongosh
> show dbs
SD      0.000GB
admin   0.000GB
config  0.000GB
local   0.000GB
> use SD
switched to db SD
> show collections
familia
mascotas
> db.familia.find()
{"_id": ObjectId("5ad64c0db6809c6a8dbb0fb8"), "tipo": "Hermano", "nombre": "Pepe", "edad": 46}
{"_id": ObjectId("5ad64ce4b6809c6a8dbb0fb9"), "tipo": "cuñado", "nombre": "Juan", "edad": 47, "comida": "Cocido"}
{"_id": ObjectId("5ad64cf2b6809c6a8dbb0fba"), "tipo": "Hermano", "edad": 47}
{"_id": ObjectId("5ad64d7d48d6016aa208f8c"), "tipo": "Hermano", "nombre": "Antonio", "afición": "TV", "edad": 50}
> db.mascotas.find()
{"_id": ObjectId("5ad65508f6efc76d03ddb6f"), "tipo": "perro", "raza": "galgo", "nombre": "Boby"}
{"_id": ObjectId("5ad6551ff6efc76d03ddb70"), "tipo": "gato", "raza": "siames", "nombre": "Cathy"}
>
```

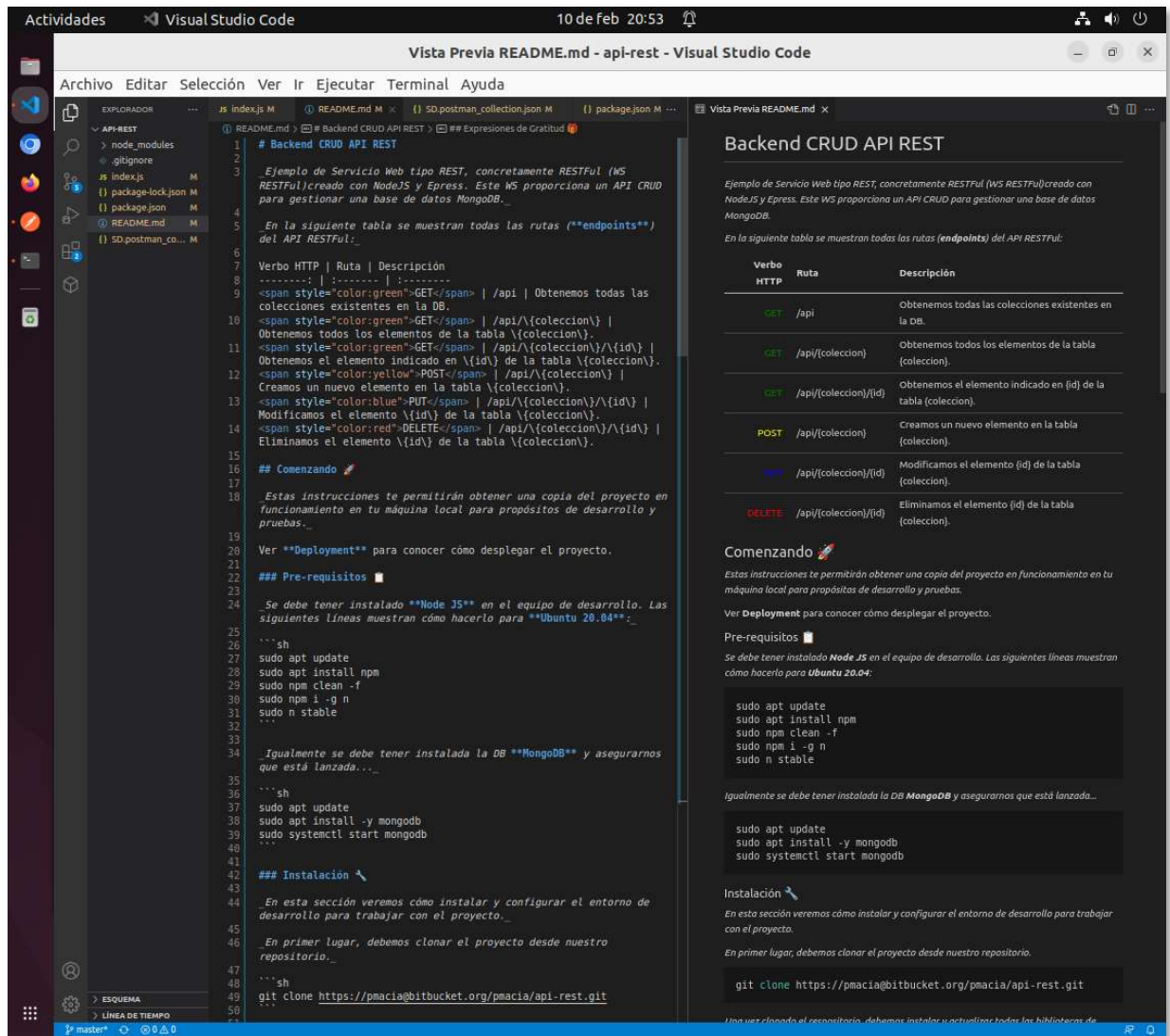
Una vez probadas todas las rutas y guardadas en una colección de **Postman**, las exportaremos a un archivo **JSON** que copiaremos en la carpeta de nuestro proyecto, sustituyendo al existente (por ejemplo, `crud.postman_collection.json`).

Veamos un último ejemplo en el que se muestra la traza creada en la terminal de nuestra aplicación (generada por **Morgan**) y una captura de una sesión con el cliente **mongosh** en la que se muestra un conjunto de consultas sobre la base de datos que se ha generado a partir de la traza mostrada.

```
<Ctrl+Alt+T>
~/node/api-crud$ npm start
[nodemon] starting `node index.js`
API REST ejecutándose en http://localhost:3000/api/:coleccion/:id
POST /api/coche 200 193.089 ms - 94
POST /api/coche 200 2.459 ms - 101
POST /api/coche 200 2.204 ms - 96
GET /api/coche 200 32.941 ms - 295
GET /api/coche/62169803598a471244c9f4ef 200 2.528 ms - 96
PUT /api/coche/62169803598a471244c9f4ef 200 2.856 ms - 28
GET /api/coche/62169803598a471244c9f4ef 200 2.089 ms - 119
PUT /api/coche/62169803598a471244c9f4ef 200 1.689 ms - 28
DELETE /api/coche/62169803598a471244c9f4ef 200 2.828 ms - 31
GET /api/coche/62169803598a471244c9f4ef 200 1.667 ms - 4
POST /api/coche 200 2.115 ms - 96
GET /api 200 17.855 ms - 9
POST /api/moto 200 55.482 ms - 95
POST /api/moto 200 1.756 ms - 96
POST /api/moto 200 1.656 ms - 95
GET /api 200 1.447 ms - 16
^C
~/node/api-crud$
```

```
<Ctrl+Alt+T>
$ mongosh
MongoDB shell version v3.6.8
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.8
> show dbs
SD      0.000GB
admin   0.000GB
config  0.000GB
local   0.000GB
> use SD
switched to db SD
> show collections
coche
moto
> db.coche.find()
{ "_id" : ObjectId("6216979c598a471244c9f4ed"), "marca" : "BMW", "modelo" : "320D", "color" : "negro", "año" : "2011" }
{ "_id" : ObjectId("621697f7598a471244c9f4ee"), "marca" : "Hyundai", "modelo" : "tucson", "color" : "blanco", "año" : "2010" }
{ "_id" : ObjectId("62169918598a471244c9f4f0"), "marca" : "Hyundai", "modelo" : "i20", "color" : "rojo", "año" : "2015" }
> db.moto.find()
{ "_id" : ObjectId("62169953598a471244c9f4f1"), "marca" : "Yamaha", "modelo" : "250", "color" : "rojo", "año" : "2018" }
{ "_id" : ObjectId("6216995f598a471244c9f4f2"), "marca" : "Yamaha", "modelo" : "250", "color" : "negra", "año" : "2018" }
> db.moto.find().pretty()
{
  "_id" : ObjectId("62169953598a471244c9f4f1"),
  "marca" : "Yamaha",
  "modelo" : "250",
  "color" : "rojo",
  "año" : "2018"
}
{
  "_id" : ObjectId("6216995f598a471244c9f4f2"),
  "marca" : "Yamaha",
  "modelo" : "250",
  "color" : "negra",
  "año" : "2018"
}
>
```

Antes de finalizar, se debe actualizar la documentación editando el archivo [README.md](#).



En la anterior imagen se muestra un ejemplo del archivo `README.md`, editado con `VSCode` en el que se aprecian dos paneles: uno de ellos con el código fuente, y el otro con una vista previa.

Para poder editar el archivo, se proporciona una plantilla y se recomienda estudiar el formato `.md` para entender cómo se da formato al texto: títulos, subtítulos, código fuente, etc.

En la siguiente figura se muestra una captura del repositorio en `Bitbucket` donde se puede apreciar cómo se visualiza el archivo `README.md` y cómo sirve de documentación del proyecto.

The screenshot shows a GitHub repository named 'api-rest'. The left sidebar contains navigation links: Fuente, Anotaciones, Ramas, Pull requests, Pipelines, Deployments, Jira issues, Seguridad, Descargas, and Repository settings. The main content area displays the README.md file. At the top, it lists repository files: index.js (2.45 KB), package-lock.json (97.52 KB), and package.json (857 B), all updated 5 minutes ago. The README content includes a title 'Backend CRUD API REST', a description of the project as a RESTful API for MongoDB, a table of endpoints, and sections for 'Comenzando' and 'Pre-requisitos'.

**Backend CRUD API REST**

Ejemplo de Servicio Web tipo REST, concretamente RESTful (WS RESTful) creado con NodeJS y Epress. Este WS proporciona un API CRUD para gestionar una base de datos MongoDB.

En la siguiente tabla se muestran todas las rutas (**endpoints**) del API RESTful:

Verbo HTTP	Ruta	Descripción
GET	/api	Obtenemos todas las colecciones existentes en la DB.
GET	/api/{coleccion}	Obtenemos todos los elementos de la tabla {coleccion}.
GET	/api/{coleccion}/{id}	Obtenemos el elemento indicado en {id} de la tabla {coleccion}.
POST	/api/{coleccion}	Creamos un nuevo elemento en la tabla {coleccion}.
PUT	/api/{coleccion}/{id}	Modificamos el elemento {id} de la tabla {coleccion}.
DELETE	/api/{coleccion}/{id}	Eliminamos el elemento {id} de la tabla {coleccion}.

**Comenzando**

Estas instrucciones te permitirán obtener una copia del proyecto en funcionamiento en tu máquina local para propósitos de desarrollo y pruebas.

Ver **Deployment** para conocer cómo desplegar el proyecto.

**Pre-requisitos**

Se debe tener instalado **Node JS** en el equipo de desarrollo. Las siguientes líneas muestran cómo hacerlo para **Ubuntu 20.04**:

```
sudo apt update
sudo apt install npm
```

**Nota:** antes de continuar, recordemos modificar la versión del código a **v3.0.0** (editando el archivo `package.json`) y subir una copia etiquetada al repositorio remoto:

```
$ git add .
$ git commit -m "API REST CRUD con MongoDB"
$ git push
$ git tag v3.0.0
$ git push --tags
```