

# API REST: Node.js

Incluya rápidamente operaciones CRUD, manejo de errores, clasificación y filtrado en su API REST hoy

## Configuración del entorno

¡Comencemos! Primero, déjame revisemos las herramientas que usaremos:

- `Node.js`: Un entorno de tiempo de ejecución JavaScript de código abierto que le permite ejecutar código fuera de un navegador. Desarrollaremos nuestra API RESTful en JavaScript en un servidor Node.js
- `MongoDB`: la base de datos en la que escribiremos nuestros datos.
- `Postman`: la herramienta que usaremos para probar nuestra AP.
- `VSCode`: puedes usar el editor de texto que quieras. Para el ejercicio usaremos VSCode.

## Introducción a REST API

Hoy en día se escucha sobre REST API en todas partes en el mundo tecnológico actual, pero ¿qué es? Para empezar, API significa Interfaz de Programación de Aplicaciones.

¿Cuál es el beneficio de una API? Permite que dos piezas de software se comuniquen entre sí. Existen muchos tipos de API: SOAP, XML-RPC, JSON-RPC, pero en este artículo hablaremos de REST.

¿Qué es REST? Es sinónimo de REpresentational State Transfer (Transferencia de Estado de Representación). Es un estilo de arquitectura de software que se utiliza para crear servicios web. REST ha facilitado que los sistemas informáticos se comuniquen entre sí a través de Internet.

¿Como funciona? Bastante similar a cómo escribe «rest api» en Google y se devuelven los resultados de búsqueda. Explicándolo de la forma más sencilla, un cliente realiza una llamada (solicitud) en forma de URL a un servidor que solicita algunos datos. A continuación se muestra un ejemplo de una búsqueda en Google para «API REST»

`www.google.com/search?q=rest+api`

El servidor luego responde con los datos (respuesta) a través del protocolo HTTP. Estos datos aparecen en notación JSON y se convierten en una imagen estéticamente agradable en su navegador: los resultados de búsqueda de Google.

En resumen, envía una solicitud en forma de URL y recibe una respuesta en forma de datos.

¿Cómo se reciben estos datos normalmente? Generalmente en formato JSON. JSON (Javascript Object Notation) describe sus datos en pares clave-valor. JSON facilita la lectura de los datos tanto en máquinas como en humanos.

## Requests

Primero, es importante comprender que las solicitudes constan de cuatro partes.

Endpoint: la URL que solicita. Típicamente compuesto por una raíz y una ruta. P.ej. <https://www.google.com/search?q=rest+api>, donde la raíz es `https://www.google.com` y la ruta es `/ search? q = rest + api`.

Método: el tipo de solicitud que envía al servidor o «verbo HTTP». Así es como podemos ejecutar nuestras operaciones CRUD (Crear, Leer, Actualizar o Eliminar). Los cinco tipos son GET, POST, PUT, PATCH y DELETE. (fuente: <https://developer.mozilla.org/es/docs/Web/HTTP/Methods>)

Encabezados: información adicional que se puede enviar al cliente o servidor que se utiliza para ayudar a sus datos de alguna manera. Por ejemplo, se puede usar para autenticar a un usuario para que puedan ver los datos. O puede decirle cómo se deben recibir los datos (aplicación / JSON).

Datos (body): la información que queremos recibir de nuestra solicitud como JSON.

Revisando con mas detalle los endpoints: dada la importancia de estos, estos son los conceptos adicionales que debemos manejar:

- Dos puntos (:) Se usa para indicar una variable en la cadena. En API REST, verá endpoints como `:username` (o algo similar). Solo sepa que cuando lo pruebe, debe reemplazar ese nombre de usuario con un nombre de usuario real. Por ejemplo:

```
prueba.com/users/:username/articles?query=value&query2=value2
```

`prueba.com/users/epadilla/articles?published=true&date=today`

- Interrogación de cierre (?): Comienza los parámetros de consulta. Los parámetros de consulta son conjuntos de pares clave-valor que se pueden usar para modificar sus solicitudes. Aquí hay un ejemplo de un endpoint en el que quiero ver mis artículos y si han sido publicados o no:

`prueba.com/users/:username/articles?query=value`

`prueba.com/users/epadilla/articles?published=true`

- Ampersand (&): Se utiliza para separar los parámetros de consulta cuando desea usar múltiples. Por ejemplo, podemos ver mis artículos publicados y publicados hoy

`prueba.com/users/:username/articles?query=value&query2=value2`

`prueba.com/users/epadilla/articles?published=true&date=today`

Una API REST debe tener al menos las siguientes cuatro partes:

- Servidor: se utiliza para establecer todas las conexiones que necesitamos, así como para definir información importante como puntos finales, puertos y enrutamiento.
- Modelo: ¿Cómo son nuestros datos?
- Rutas: ¿A dónde van nuestros endpoint?
- Controladores: ¿Qué hacen nuestros endpoints?

### Configurando el Server:

Empecemos agregando las dependencias:

```
npm install mongoose
```

```
npm install body-parser
```

```
npm install express
```

Después de instalar las dependencias su `package.json` debe lucir similar al siguiente:

```
HellowWorld > {} package.json > ...
1  {
2    "name": "hellowworld",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "author": "",
10   "license": "ISC",
11   "dependencies": {
12     "body-parser": "^1.19.0",
13     "express": "^4.17.1",
14     "mongodb": "^3.5.4",
15     "mongoose": "^5.9.2"
16   }
17 }
18 |
```

Ahora sí, empezamos por el `main.js`. sustituimos el código, por el siguiente:

```
var express = require("express"),
    app = express(),
    port = process.env.PORT || 3000,
    mongoose = require("mongoose"),
    bodyParser = require("body-parser"),
    Entry = require("./api/models/leaderboardModel");
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

mongoose.connect(

"mongodb+srv://ryangleason:leaderboard@cluster0-6mky3.mongodb.net/
RESTTutorial?retryWrites=true&w=majority",
  );

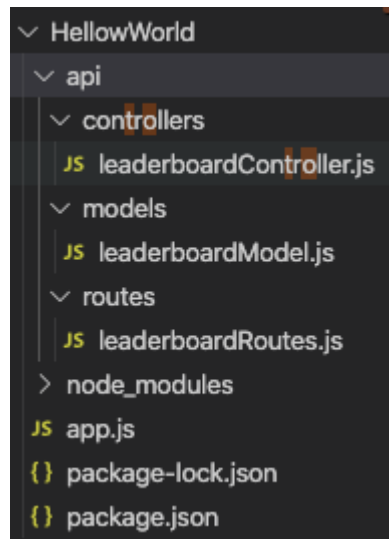
var routes = require("./api/routes/leaderboardRoutes");
```

```
routes (app) ;  
  
app.listen (port) ;  
  
console.log ("API server started on " + port) ;
```

Explicando un poco este archivo:

- **Express**: el marco de la aplicación web para `Node.js`. Esto inicia un servidor y escucha las conexiones en el puerto 3000.
- **Mongoose**: nos ayuda a modelar objetos para MongoDB. Ayuda a gestionar las relaciones entre los datos, valida el esquema y, en general, es útil para comunicarse desde `Node.js` a MongoDB.
- **bodyParser**: es necesario para interpretar las solicitudes. Funciona extrayendo la parte del cuerpo de una solicitud entrante y la expone como `req.body` y como JSON.
- **Entry**: este es el nombre de nuestro esquema. Necesitamos incluir esto en nuestro `app.js` porque el esquema debe estar «registrado» para cada modelo que usemos.
- `mongoose.connect (cadena uri, {opciones})`: establece una conexión con nuestra base de datos Atlas MongoDB. Establecemos estas opciones para eliminar las advertencias de desuso.
- `routes (app)`: – Define cómo responderán los `endpoints` de una aplicación a las solicitudes de los clientes. Esto apuntará a las rutas que nos definimos.

Antes de poder probar nuestro REST API, necesitamos crear nuestro modelo, rutas y el controlador. Así es como se ve la estructura de directorios del proyecto:



## Creando el modelo

```
var mongoose = require("mongoose");
var Schema = mongoose.Schema;

var EntrySchema = new Schema({
  player: {
    type: String,
    required: "Please enter a name"
  },
  score: {
    type: Number,
    required: "Please enter a score"
  },
  registered: {
    type: String,
    default: "No"
  }
})
```



```
});  
  
module.exports = mongoose.model("Entry", EntrySchema,  
"Leaderboard");
```

Se ha creado un modelo básico, que solo contiene tres campos. Es importante comprender que estamos utilizando un esquema Mongoose. Cada esquema se asigna a una colección MongoDB y define la estructura de los documentos dentro de esa colección. Luego verá que al final estamos exportando nuestro esquema, llamado `Entry`, y exponiéndolo al resto de nuestra aplicación.

Cada `Entry` representará un registro diferente en nuestra «tabla» de clasificación donde una `Entry` es el nombre del jugador, el puntaje y si están registrados.

## Definiendo Rutas

```
module.exports = app => {  
  var leaderboard =  
  require("../controllers/leaderboardController");  
  
  app  
    .route("/entries")  
    .get(leaderboard.read_entries)  
    .post(leaderboard.create_entry);  
  
  app  
    .route("/entries/:entryId")  
    .get(leaderboard.read_entry)  
    .put(leaderboard.update_entry)  
    .delete(leaderboard.delete_entry);  
};
```

Las rutas definirán lo que sucede cuando un usuario llega a uno de nuestros endpoints. También es cómo determinamos qué operación CRUD se utiliza para interactuar con los datos en nuestra base de datos.

- POST — Create un entry
- GET — Read un entry
- PUT — Update un entry
- DELETE — Delete un entry

## Definamos el Controlador

```
var mongoose = require("mongoose"),
    Entry = mongoose.model("Entry");

exports.read_entries = async (req, res) => {
  ret = await Entry.find();
  res.json(ret);
};

exports.create_entry = async (req, res) => {
  const new_entry = new Entry(req.query);
  ret = await new_entry.save();
  res.json(ret);
};

exports.read_entry = async (req, res) => {
  ret = await Entry.findById(req.params.entryId);
  res.send(ret);
};
```

```
exports.update_entry = async (req, res) => {
  ret = await Entry.findByIdAndUpdate({ _id: req.params.entryId },
  req.body, {
    new: true
  });
  res.json(ret);
};

exports.delete_entry = async (req, res) => {
  await Entry.deleteOne({ _id: req.params.entryId });
  res.json({ message: "Entry deleted" });
};
```

Aquí es donde «sucede la magia». Entonces, ¿qué está sucediendo cuando nuestras rutas se ven afectadas? Este es el mínimo necesario para una API completamente funcional. Cada uno de estos métodos corresponde con una ruta: notará que todos los nombres son los mismos que las rutas.

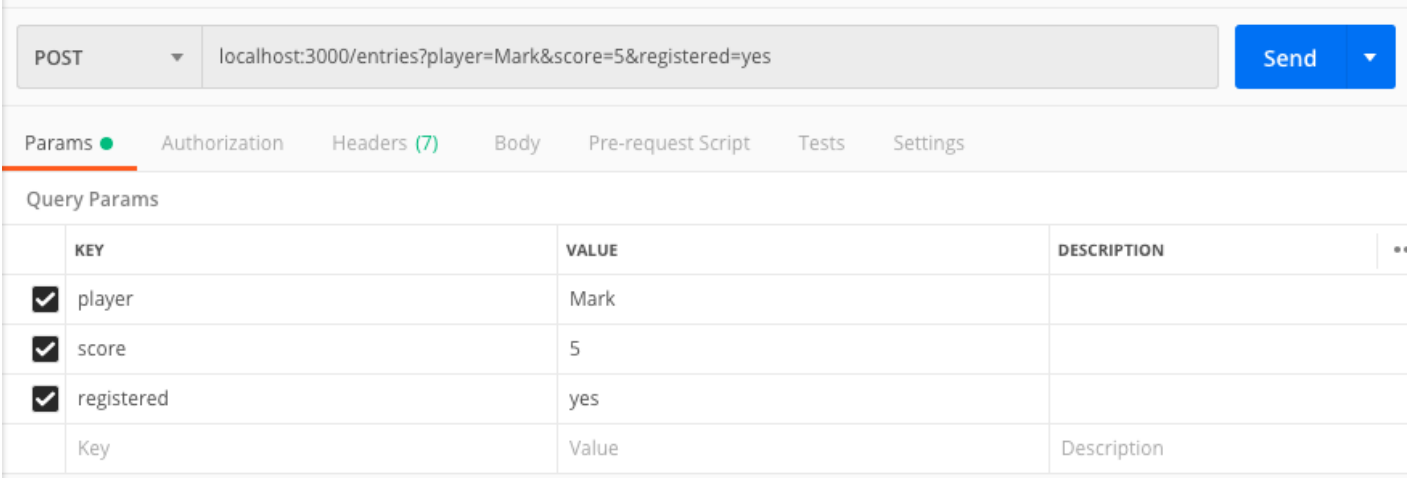
Cada uno de estos métodos incluye dos parámetros: la solicitud y la respuesta (`req, res`). Luego usamos el método Mongoose que coincide con la operación que estamos tratando de implementar. Por ejemplo, para una solicitud `GET`, queremos encontrar todos los documentos y devolverlos como un objeto `JSON` de respuesta. Para hacerlo, utilizamos el método `find()`.

Lo mismo vale para `POST`. Queremos crear un registro, por lo que utilizamos el método Mongoose, `save()`, para escribir ese registro en la base de datos.

Ha llegado el momento de probar nuestra aplicación.

```
Elmers-MacBook-Air:HellowWorld elmer$ node app.js  
API server started on 3000  
█
```

## Creemos nuestro primer registro, usando postman



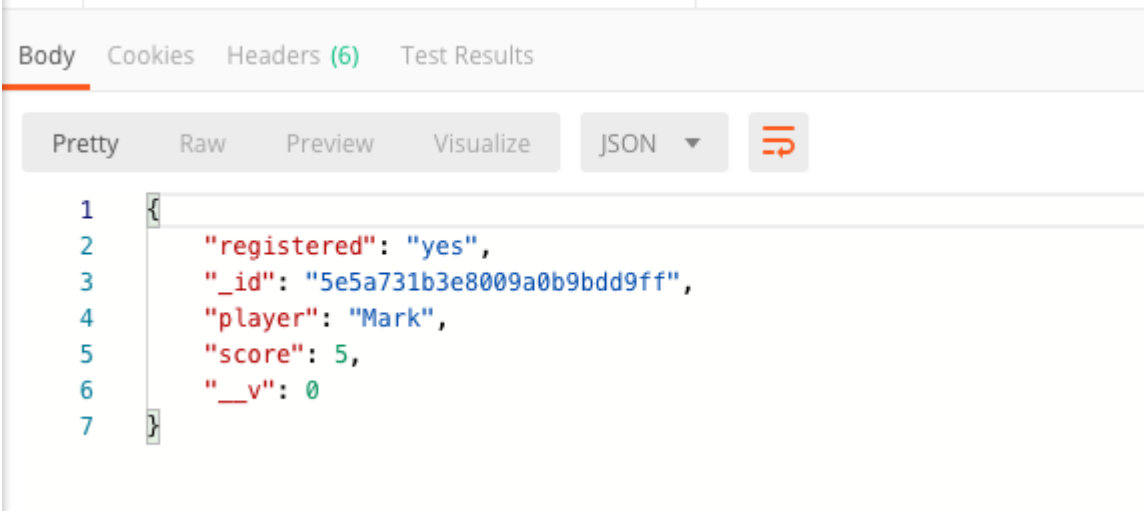
POST localhost:3000/entries?player=Mark&score=5&registered=yes Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	player	Mark	
<input checked="" type="checkbox"/>	score	5	
<input checked="" type="checkbox"/>	registered	yes	
	Key	Value	Description

Así es como se verá una solicitud POST enviada a `localhost:3000/Entries`. Aquí está la respuesta que debe obtener al presionar el botón Send:



Body Cookies Headers (6) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "registered": "yes",  
3   "_id": "5e5a731b3e8009a0b9bdd9ff",  
4   "player": "Mark",  
5   "score": 5,  
6   "__v": 0  
7 }
```

Escuela ORT

Orientación: Informática

Materia: DAI



¡Felicidades! Acaba de ingresar su primer registro en la base de datos. Puede confirmarlo consultando las Collections en MongoDB Atlas:

The screenshot shows the MongoDB Atlas web interface. The top navigation bar includes the MongoDB Atlas logo, 'All Clusters', and a warning icon with the text 'Please set your time zone' and 'Usage'. The left sidebar contains a 'CONTEXT' section with a dropdown for 'Project 0', and a navigation menu with categories: ATLAS (Clusters, Data Lake BETA), SECURITY (Database Access, Network Access, Advanced), PROJECT (Access Management, Activity Feed, Alerts, Integrations, Settings), SERVICES (Charts, Stitch, Triggers), and HELP (Docs). The main content area is titled 'ELMER'S ORG - 2020-02-29 > PROJECT 0 > CLUSTERS' and shows 'Cluster0'. Below this, there are tabs for Overview, Real Time, Metrics, Collections (selected), Profiler, Performance Advisor, and Command Line Tools. The 'Collections' tab shows 'DATABASES: 1' and 'COLLECTIONS: 1'. A '+ Create Database' button is visible. Below it, a search bar for 'NAMESPACES' shows a dropdown with 'test' and 'Leaderboard'. The 'test.Leaderboard' collection is selected, showing 'COLLECTION SIZE: 79B', 'TOTAL DOCUMENTS: 1', and 'INDEXES TOTAL SIZE: 20KB'. There are tabs for Find (selected), Indexes, Aggregation, and Search BETA. A 'FILTER' button with the text '{\"filter\": \"example\"}' is present. Below the filter, it says 'QUERY RESULTS 1-1 OF 1'. The query result is a JSON document: 

```
{  \"_id\": ObjectId(\"5e5a731b3e8009a0b9bdd9ff\"),  \"registered\": \"yes\",  \"player\": \"Mark\",  \"score\": 5,  \"_v\": 0}
```