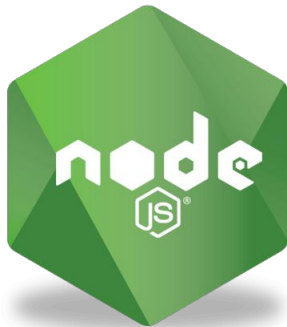




# Node Rest API con Express y SQLite





## Requisitos para este proyecto:

- ☐ Node.js instalado en el computador: <https://nodejs.org/es/>
- ☐ Un IDE (Entorno de Desarrollo Integrado) que soporte JavaScript y SQLite 3.
- ☐ DB Browser o cualquier herramienta de gestión y visualización de bases de datos compatible con SQLite 3.
- ☐ Postman, o cualquier plataforma para control y prueba de APIs.
- ☐ Conocimiento básico de JavaScript.
- ☐ Conocimiento básico de bases de datos.





## Requisitos Opcionales:

- ☐ Leer la documentación de Node.js (disponible en español).
- ☐ Leer la documentación de Express.js (disponible en español).
- ☐ Leer la documentación de SQLite.





## Qué es Node.js?

- ❑ Node.js es un framework de código abierto, multiplataforma y que le permite al desarrollador ejecutar el código JavaScript por fuera del navegador. Y es precisamente la necesidad de ejecutar este lenguaje del lado del servidor el que hace surgir a este framework.
- ❑ Node.js no solo permite crear sitios web interactivos, sino que los hace más ágiles y capaces de trabajar con otros lenguajes de secuencia como Python.





## Qué es Express.js?

- ❑ Express.js es un framework (entorno de trabajo en español), que “escucha” requerimientos (requests) por parte del usuario y responde a través de texto (response).
- ❑ Toda la Web está basada en requerimientos por parte del usuario y respuestas por parte del servidor.





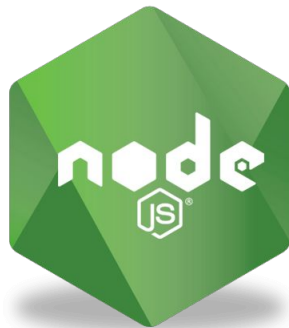
## Qué es SQLite?

- ❑ Formada por una biblioteca en lenguaje C, SQLite es una base de datos relacional muy utilizada por desarrolladores en todo el mundo. Funciona como un servidor propio e independiente, ya que el SGBD (Sistema Gestor de Bases de Datos) se puede ejecutar de manera muy fácil, eliminando así consultas y procesos por separado.
- ❑ La biblioteca SQLite se genera y almacena directamente en el archivo de la base de datos.





# Creación del Archivo Base





## Creación del archivo base: Primeros pasos (Terminal)

1. En Terminal, crear un nuevo fichero con el nombre del proyecto (cwc-cars) y acceder al mismo:

```
% mkdir cwc-cars
```

```
% cd cwc-cars
```

2. Iniciar un nuevo repositorio local con git en el nuevo fichero (buenas prácticas).

```
% git init
```

3. Crear un archivo .gitignore y agregar node\_modules al archivo. Guardar y salir.

```
% vi .gitignore
```

```
node modules
```







## Creación del archivo base: Primeros pasos (Terminal)

4. Ejecutar el comando `npm init -y` . Esto activará la inicialización del proyecto.

```
% npm init -y
```

5. Ejecutar el comando `npm i express`. Esto instalará la dependencia Express en el fichero.

```
% npm i express
```

6. Acceder al IDE.





## Creación del archivo base: Primeros pasos (Terminal)

- ❑ Al abrir el fichero <cwc-cars> en el entorno de desarrollo, debería haber los siguientes archivos ya creados (esto indica que los anteriores pasos fueron ejecutados exitosamente).

1. Fichero node\_modules
2. Archivo .gitignore
3. Archivo package-lock.json
4. Archivo package.json





# Creación del Servidor





## Creación de la Estructura Básica de un Servidor

- ❑ En la carpeta maestra (cwc-cars), crear un archivo con nombre “server.js”, con el siguiente bloque de código:

```
const express = require("express"); // Referencia a la función Express del framework.
const app = express(); // Se instancia el objeto Express y se asigna a una variable.

app.get("/test", (req, res) =>{ // Cuando el navegador pide información a través de '/test'
    res.status(200).json({ success: true}); // Responde que hubo éxito en el requerimiento
});

app.listen(1337, ()=> console.log("server is running on port 1337"));
// Abre un canal de escucha por el puerto 1337 e imprime por consola el resultado.
```





## Prueba del Correcto Funcionamiento del Servidor

❑ Para probar el correcto funcionamiento del simulador de servidor web, se debe:

1. Abrir una ventana del Terminal en el fichero <cwc-cars> y escribir el comando

```
% node server.js
```

(Debería aparecer en consola el mensaje “server is running in port 1337”).

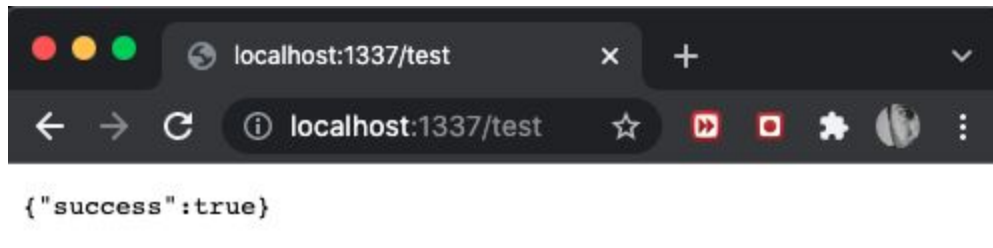
2. Abrir una ventana nueva de un navegador y en la barra escribir: localhost:1337/test





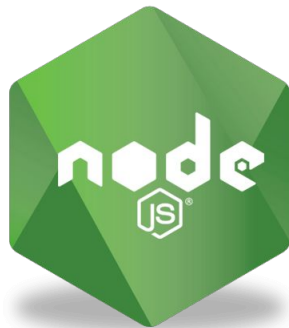
## Prueba del Correcto Funcionamiento del Servidor

- ❏ En el navegador debería aparecer:





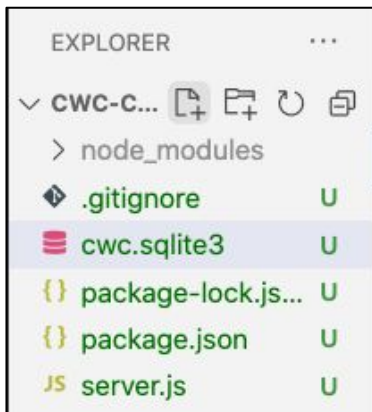
# Configuración de la Base de Datos:





## Configuración de la Base de Datos:

- ❑ En el fichero maestro (cwc-cars), crear el archivo cwc.sqlite3.



- ❑ Abrir el archivo cwc.sqlite3 con DB Browser.

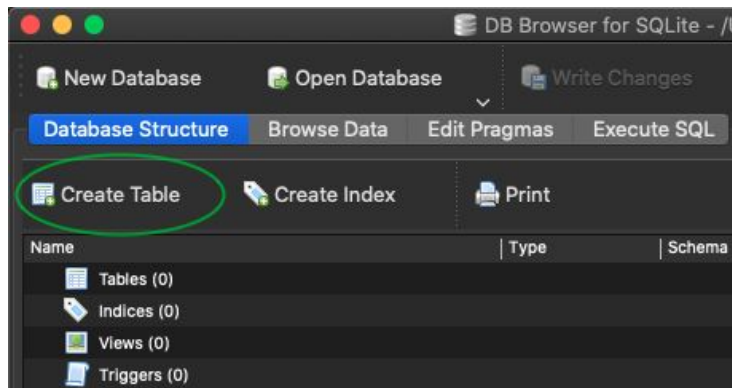






## Configuración de la Base de Datos:

- ❑ En DB Browser, oprimir el botón “Create Table”.





## Configuración de la Base de Datos:

- ❑ En la ventana emergente, nombrar la tabla “cars” y crear los campos “id”, “make”, “model” y “year”. Presionar OK.

Nombre de la tabla.

Agregar campo

Nombre del campo

Tipo de Dato

Table

**cars**

Advanced

Fields Constraints

Add Remove Move to top Move up Move down Move to bottom

Name	Type	NN	PK	AI	U	Default	Check	Co
id	INTEGER	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>			
make	TEXT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			
model	TEXT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			
year	INTEGER	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			

```
1 CREATE TABLE 'cars' (  
2   'id' INTEGER,  
3   'make' TEXT,  
4   'model' TEXT,  
5   'year' INTEGER,  
6   PRIMARY KEY('id' AUTOINCREMENT)  
7 );
```

Cancel OK

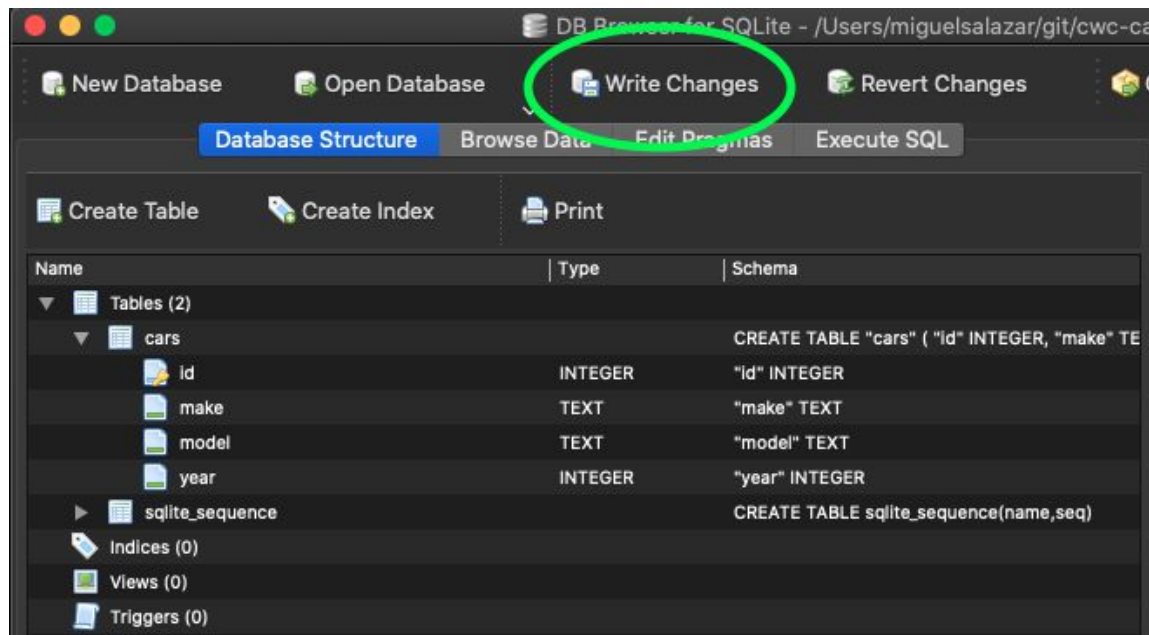
Tipos de Columnas  
(PK y AI para el id)





## Configuración de la Base de Datos:

- Una vez creada la tabla exitosamente, presionar “Write changes”.





## Configuración de la Base de Datos:

- ❏ Una vez creada la base de datos, volver a la Terminal e instalar los módulos “knex” y “sqlite3” mediante el comando:

```
% npm i knex sqlite3
```

Nota: Presionar ctrl+c antes de instalar los módulos en caso de que haya alguna ejecución en proceso.





## Configuración de la Base de Datos:

### Qué es Knex.js?

Knex.js es un generador de consultas flexible que funciona con PostgreSQL, MySQL y SQLite3.

Permite ejecutar sentencias de SQL contra la base de datos.

### Qué es sqlite3.js?

Es un controlador que adiciona la funcionalidad de conectar bases de datos SQLite a las aplicaciones Express.





## Configuración de la Base de Datos:

- ❑ A continuación, para configurar knex es necesario crear un fichero con nombre “db” (por las siglas de Base de Datos en inglés), y dentro del fichero, un archivo con nombre “knex.js”. En este archivo, escribir el siguiente bloque de código:

```
const knex = require("knex"); //Referencia al módulo Knex.js

const connectedKnex = knex({ //Se instancia el objeto knex() y se asigna a una
  variable.
  client: "sqlite3", //Se define sqlite como la librería de preferencia
  connection: {
    filename: "cwc.sqlite3" // Se conecta al archivo que contiene la base de datos.
  }
});

module.exports = connectedKnex; //Permite exportar la instancia Knex a otro archivo
```





## Configuración de la Base de Datos:

- ❑ Una vez configurada la base de datos en el proyecto, se procede a crear un archivo de JavaScript que contenga todas las funciones CRUD que le permitan a “servidor.js” interactuar con la base de datos.
- ❑ Para esto, es necesario crear dentro del fichero “db” un archivo con nombre “cars.js” con el siguiente bloque de código:





## Configuración de la Base de Datos:

❏ Archivo "cars.js":

```
const knex = require("../knex"); //Variable que recibe una instancia conectada de knex.
```

```
//Se crean las funciones básicas para operar con la base tabla de carros.
```

```
function createCar(car) {  
    return knex("cars").insert(car); // Agrega una instancia de carro a la tabla  
};
```

```
function getAllCars() {  
    return knex("cars").select("*"); // Selecciona todo en la tabla de carros.  
};
```

```
// Continúa el bloque de código en la siguiente diapositiva...
```







## Configuración de la Base de Datos:



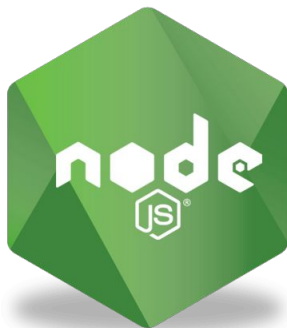
### Archivo “cars.js”:

```
function deleteCar(id){  
    return knex("cars").where("id", id).del(car); //Elimina un carro con base en el id.  
};  
  
function updateCar(id, car){  
    return knex("cars").where("id", id).update(car); // Actualiza la información de un carro.  
};  
  
module.exports = { //Permite exportar las funciones a otro archivo.  
    createCar,  
    getAllCars,  
    deleteCar,  
    updateCar  
}
```





# Configuración de Endpoints



## Configuración de endpoints en “server.js”:

- ❑ Para configurar los endpoints del servidor, se necesita volver al Terminal (ubicada en el fichero maestro) y ejecutar el siguiente comando para importar el módulo body-parser:

```
% npm i body-parser
```

- ❑ body-parser es una herramienta para el análisis gramatical de peticiones HTTP. Básicamente, lo que hace es facilitar la forma en la que se puede acceder al contenido de una petición HTTP según esté codificado.





## Configuración de endpoints en “server.js”:

- ❑ Acto seguido, se requiere volver al archivo “server.js” para configurar los endpoints, así como la configuración de body-parser y la conexión a la base de datos. Para la configuración de body parser, se agrega el siguiente bloque de código:

```
const bodyParser = require("body-parser") // 3. Referencia a la biblioteca body-parser
//Estas funciones le enseñan a Express a analizar gramaticalmente cualquier parte codificada en el URL y en json
app.use(bodyParser.urlencoded({ extended: false}));
app.use(bodyParser.json());
```



## Configuración de endpoints en “server.js”:

- Adicionalmente, se requiere configurar la conexión a la base de datos y los métodos RESTful para la API REST con el siguiente bloque de código también en el archivo “server.js”:

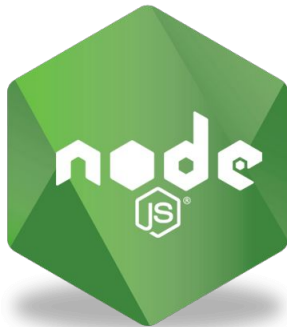
```
const db = require("../db/cars"); // Se importa una instancia del archivo cars para poder ejecutar las funciones create, delete, update y getAllcars.

//2. Se crean los endpoints necesarios para operar con la base de datos.
app.post("/cars", async (req, res) => {
  const results = await db.createCar(req.body);
  res.status(201).json({ id: results[0] });
});
app.get("/cars", async (req, res) =>{
  const cars = await db.getAllCars();
  res.status(200).json({ cars });
});
app.patch("/cars/:id", async (req, res) =>{
  const id = await db.updateCar(req.params.id, req.body);
});
app.delete("/cars/:id", async (req, res) =>{
  await db.deleteCar(req.params.id);
  res.status(200).json({ success: true});
});
```





# Prueba de Funcionamiento de la API REST con Postman





## Prueba de funcionamiento de la API REST

1. Abrir una ventana del Terminal en el fichero <cwc-cars> y escribir el comando

```
% node server.js
```

(Debería aparecer en consola el mensaje `server is running on port 1337`).

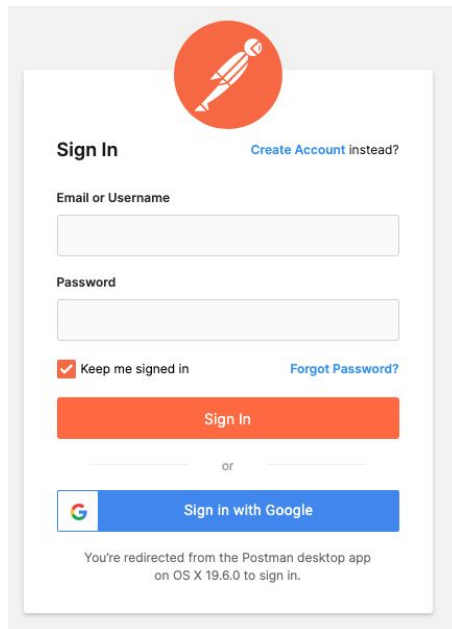
2. Abrir el programa Postman para ejecutar las pruebas.





## Prueba de funcionamiento de la API REST

1. La aplicación Postman requiere el inicio de sesión de usuario, por lo tanto es necesario bien sea crear una cuenta nueva, o iniciar sesión. El proceso es muy sencillo.



The screenshot shows the Postman Sign In page. At the top is a red circular icon with a white rocket. Below it, the text "Sign In" is followed by a link "Create Account instead?". There are two input fields: "Email or Username" and "Password". Below the password field is a checkbox labeled "Keep me signed in" which is checked, and a link "Forgot Password?". A large orange "Sign In" button is below these. Underneath is a horizontal line with "or" in the center. Below that is a "Sign in with Google" button featuring the Google logo. At the bottom, a small message states: "You're redirected from the Postman desktop app on OS X 19.6.0 to sign in."

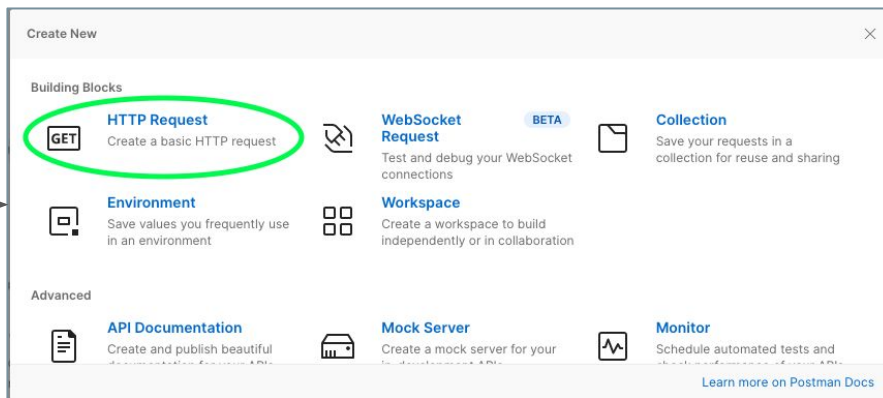
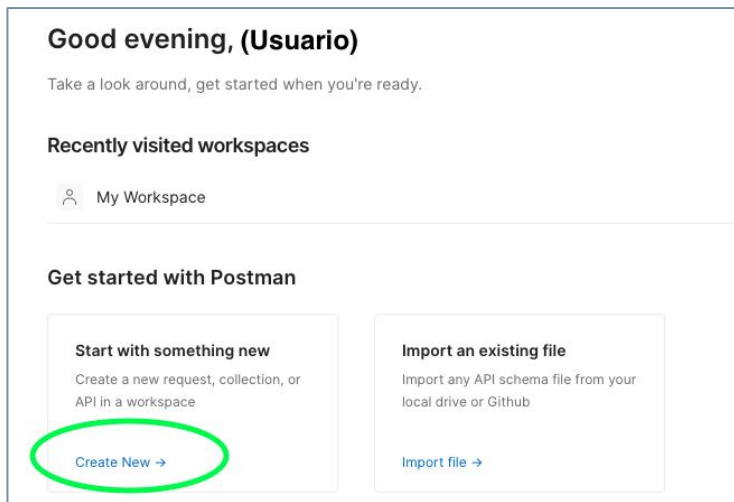






# Prueba de funcionamiento de la API REST

## 2. Una vez iniciada la sesión, crear un nuevo request.

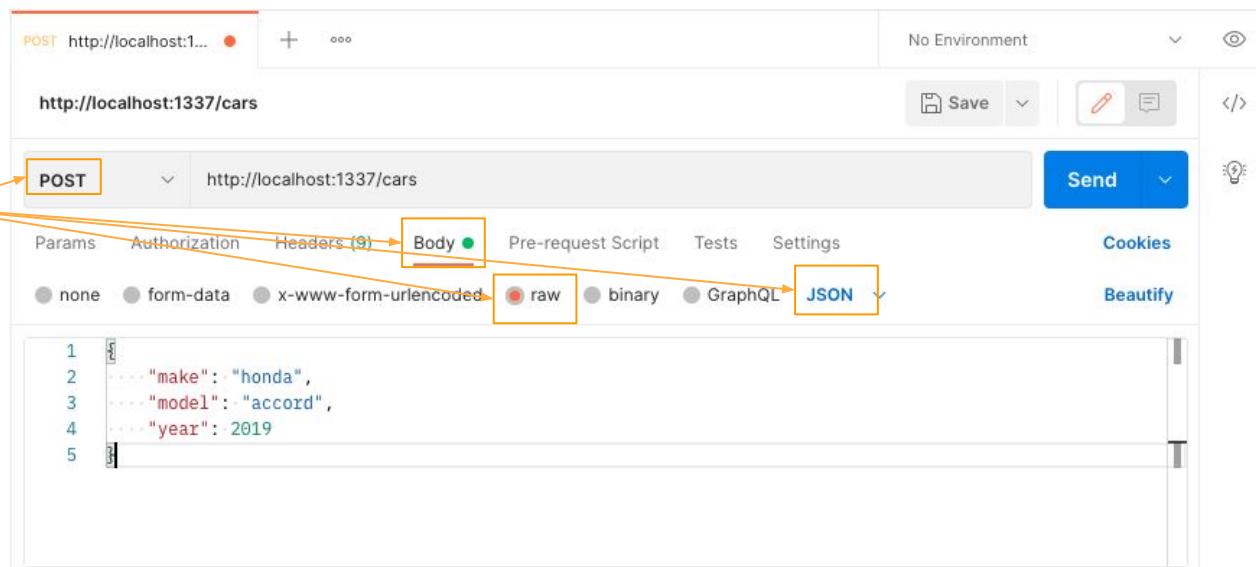




## Prueba de funcionamiento de la API REST

### 3. Seleccionar el tipo de request:

Seleccionar:  
Post  
Body  
Raw  
JSON



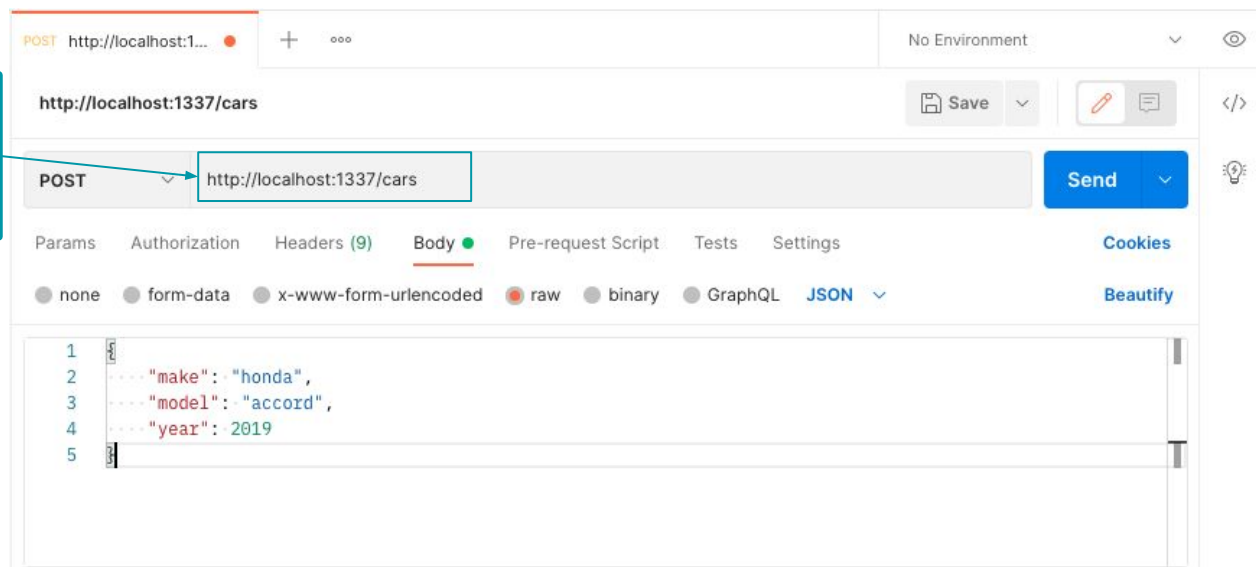


## Prueba de funcionamiento de la API REST

### 4. Ingresar la URL para enviar el request al servidor:

Ingresar URL:

http://localhost:1337/cars



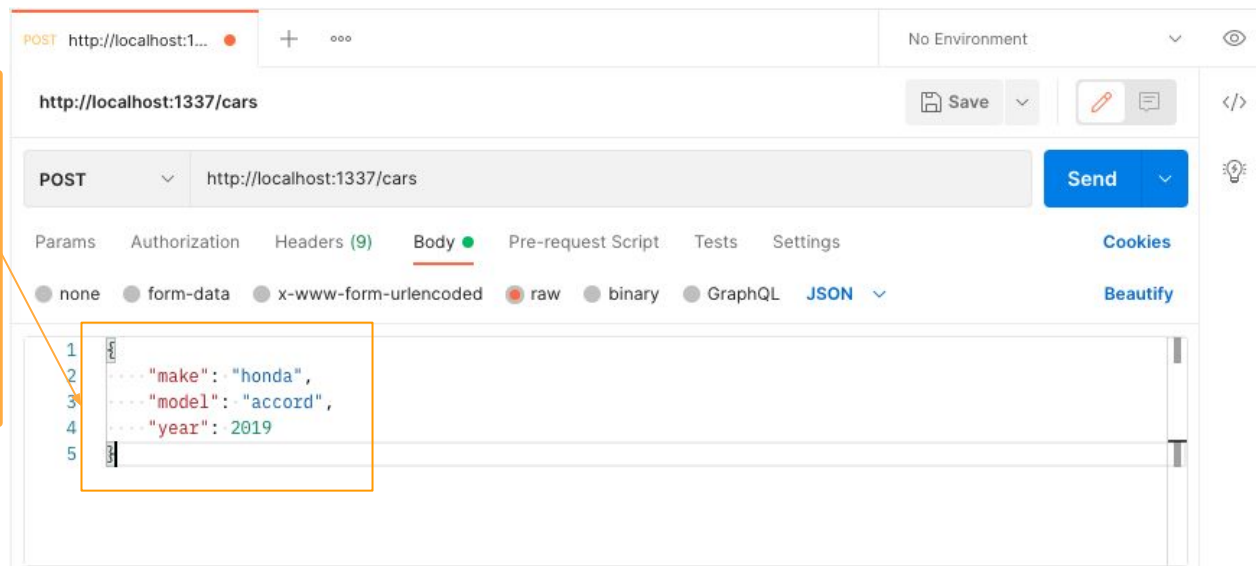


## Prueba de funcionamiento de la API REST

### 5. Ingresar el contenido del request.

Ingresar  
contenido del  
post:

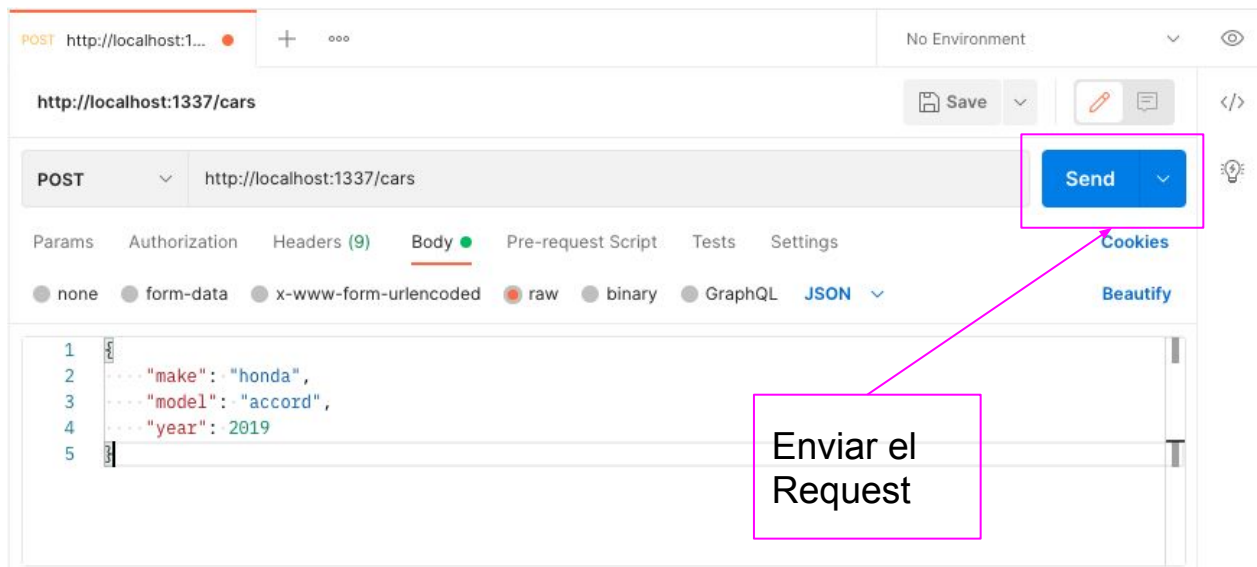
```
{  
  "make": "honda",  
  "model": "accord",  
  "year": 2019  
}
```





## Prueba de funcionamiento de la API REST

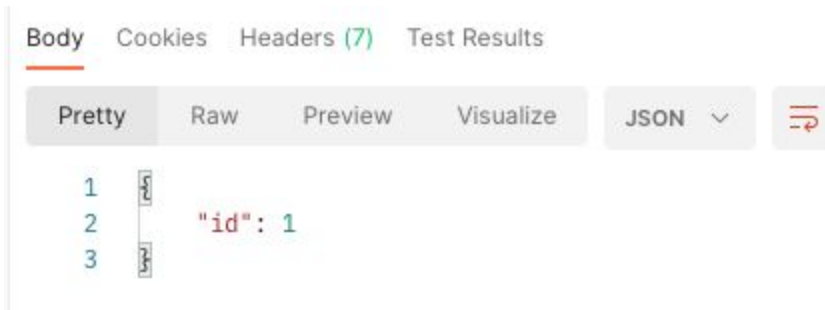
### 6. Enviar el Request.





## Prueba de funcionamiento de la API REST

Si todo está correcto, el resultado sería este:



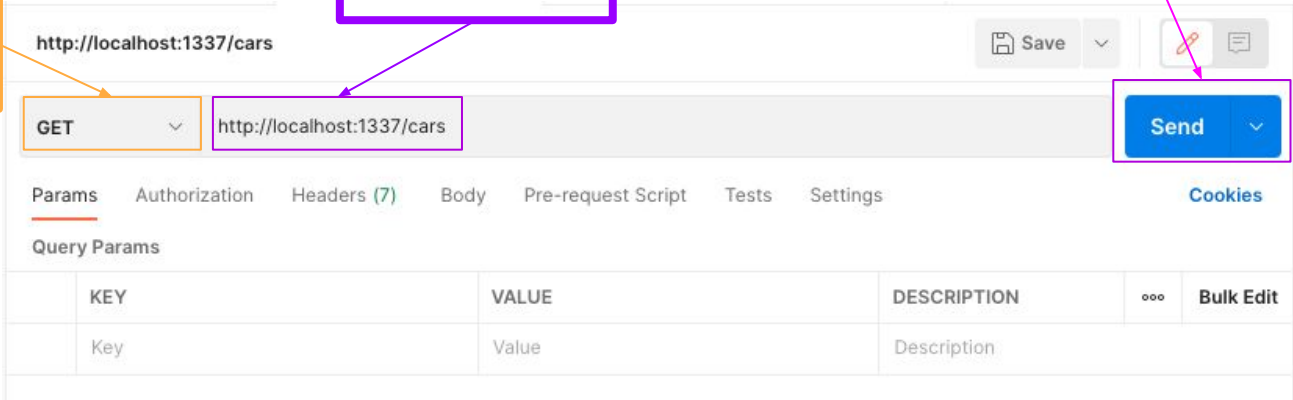
## Prueba de funcionamiento de la API REST

3. Ahora se realiza una prueba de que el carro está en la base de datos.

Selecciónar:  
GET

Misma URL

Enviar Request



http://localhost:1337/cars

GET

http://localhost:1337/cars

Send

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

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		



# Prueba de funcionamiento de la API REST

Si todo está correcto, el resultado sería este:

The screenshot shows a REST client interface with the following details:

- Body** tab is selected.
- Response status: **200 OK**, 17 ms, 298 B.
- Buttons: **Save Response** (dropdown), **Copy**, **Search**.
- Format: **JSON** (dropdown).
- Response body (Pretty):

```
1  {  
2    "cars": [  
3      {  
4        "id": 1,  
5        "make": "honda",  
6        "model": "accord",  
7        "year": 2019  
8      }  
9    ]  
1 }
```







## Repositorio:

- ❑ En dado caso de que haya alguna duda, consultar el repositorio disponible en el siguiente link y comparar los archivos y código:

[Repositorio API REST con Node y SQLite](#)

