



SOY TÉCNICO!



**5** **Acreditado**  
**años**  
En Gestión Institucional y  
Docencia de Pregrado  
hasta el 4 de Noviembre de 2026

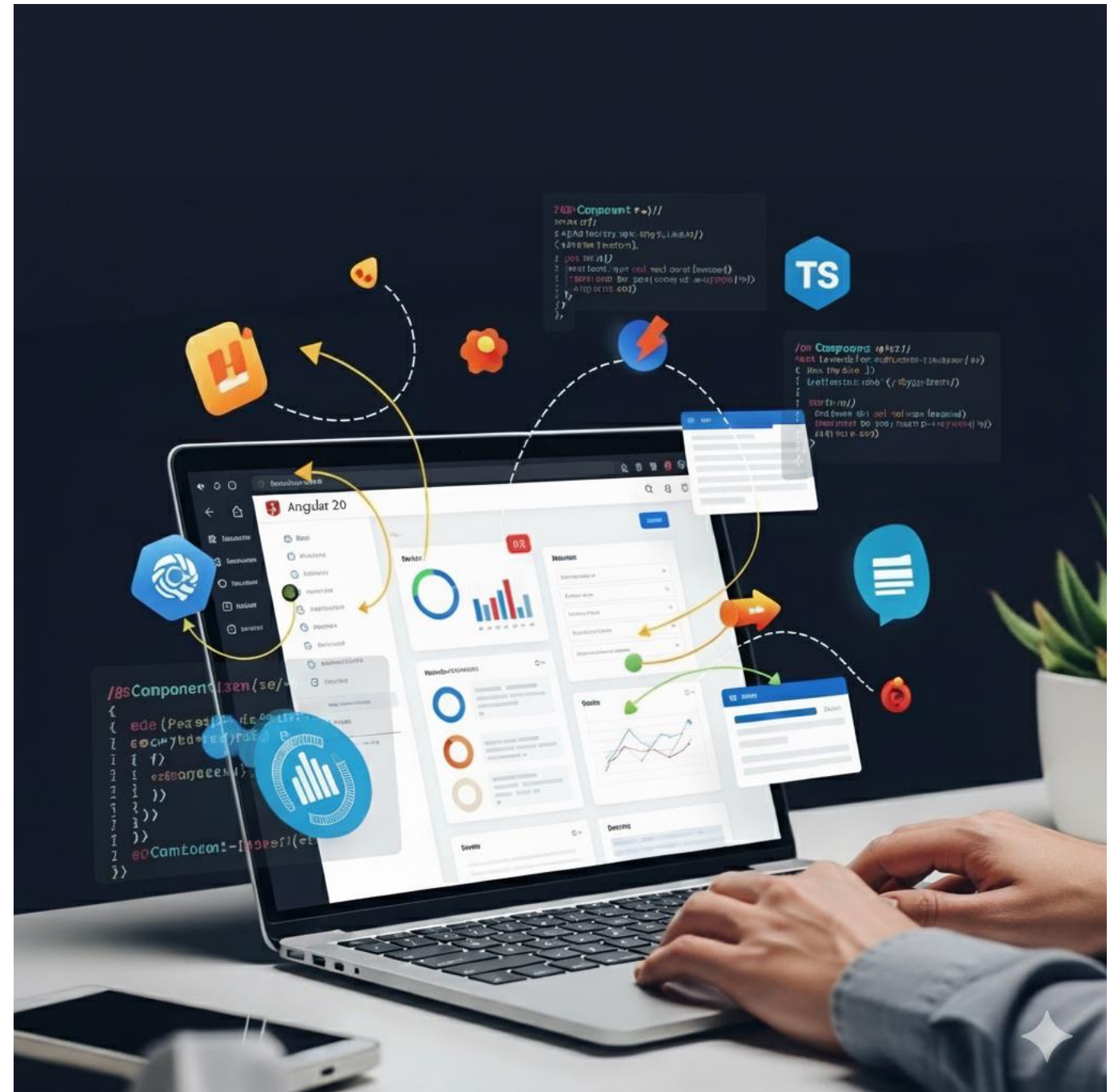
# Desarrollo de Aplicaciones Web

CEDUC UCN  
Docente: Sebastián Solar Cuevas



# Desarrollo de aplicaciones Backend

## API Rest con NodeJS y MySQL



## Contenidos a tratar:

- Entorno y requerimientos
- Elección de Ubicación
- Inicialización del proyecto
- Configuración de enrutadores (router) y navegación

# Primeros pasos: Entorno y requerimientos

- Plataformas Microsoft:
  - ✓ Microsoft Windows (10 o superior)
  - ✓ Git para Windows (uso de Git bash)
  - ✓ NodeJS ([www.nodejs.org](http://www.nodejs.org))
- Plataformas Linux / MacOS
  - ✓ Terminal
  - ✓ NodeJS

Independiente de la plataforma:

- ✓ Motor MySQL

# Primeros pasos: Elección de Ubicación

*Para las instrucciones asumiremos que estamos trabajando con Windows*

- Crea una carpeta en algún lugar que no requiera permisos de administrador (ej. Mis documentos)
- Ingresa a la carpeta y crea un archivo de título “package.json” e introduce el contenido como lo describe la imagen
- Adicionalmente crea las carpetas “middleware”, “routes”, “services” y los archivos “index.js” y “.env”
- Abre un “git batch” en la carpeta del proyecto para proceder con la instalación de los paquetes

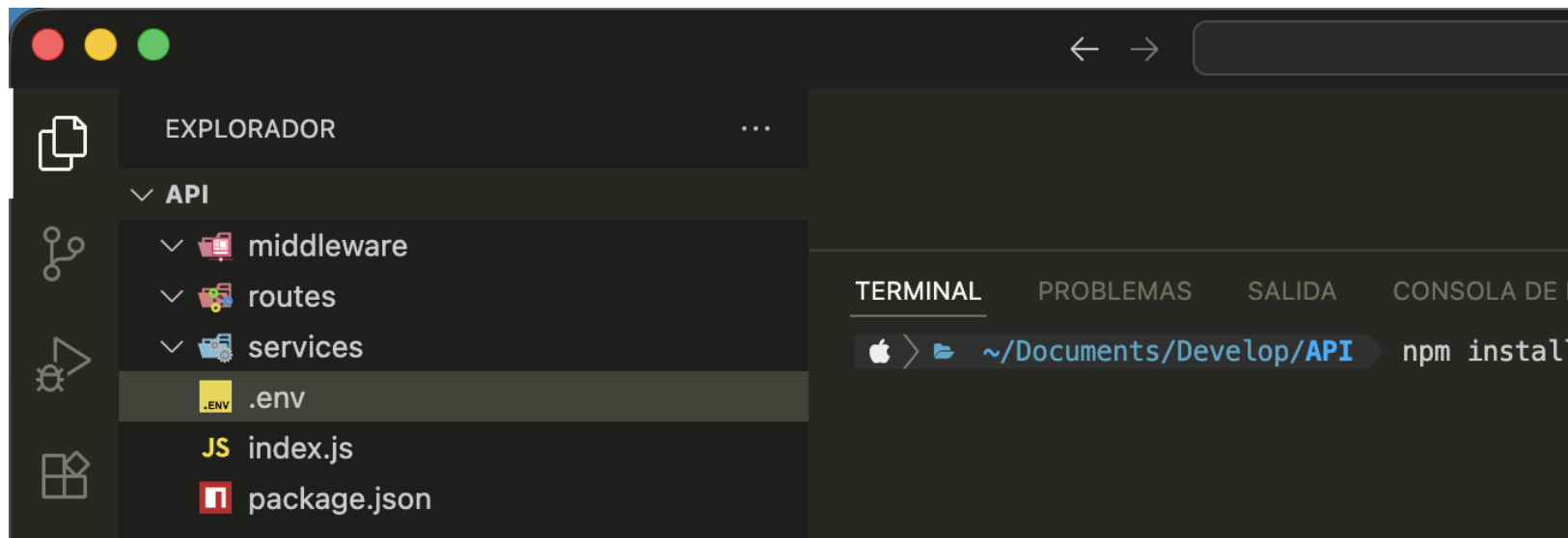
```
1  {
2    "name"       : "api",
3    "version"    : "1.0.0",
4    "description": "API REST con Node + Express",
5    "licence"    : "MIT",
6    "author"     : "Tu Nombre",
7    "type"       : "module",
8    "main"       : "index.js",
9    "scripts"    : {
10     "test"      : "echo \\Error: no test specified\\ && exit 1",
11     "start"     : "node index.js"
12   },
13   "dependencies": {
14     "cors"      : "2.8.5",
15     "dotenv"    : "^17.2.2",
16     "express"   : "^4.18.2",
17     "https"     : "^1.0.0",
18     "luxon"     : "^3.7.2",
19     "mysql"     : "^2.18.1",
20     "mysql2"    : "^3.15.0",
21     "uuid"     : "^13.0.0"
22   }
23 }
```

# Primeros pasos: Explicando los paquetes a instalar

- **dotenv:** Carga variables de entorno desde un archivo .env, como claves, constantes, configuraciones del servidor, etc.
- **express:** framework para crear servidores web, simplifica la creación de rutas, manejo de peticiones y respuestas.
- **https:** Crea un servidor web seguro (con cifrado SSL/TLS). Parte del núcleo de node pero se instala para configuraciones específicas.
- **luxon:** Librería encargada del manejo de fechas y horas
- **mysql:** Permite conectar y ejecutar consultas a BD MySQL.
- **mysql2:** Version mejorada y mas moderna de mysql (requiere la anterior para funcionar)
- **uuid:** genera identificadores únicos (UUIDs) para creación de usuarios, sesiones o recursos



# Primeros pasos: Ejecución en Terminal



```
npm install
```

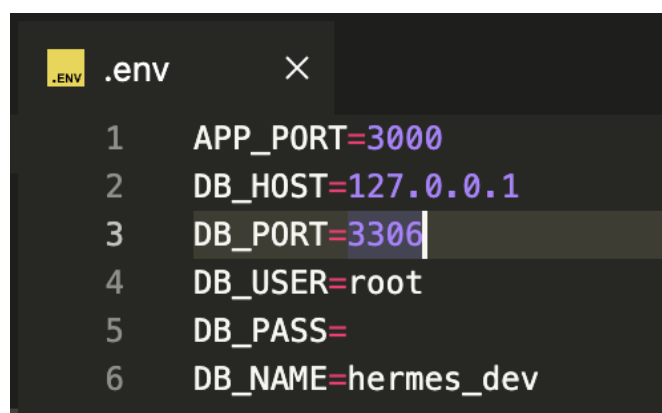
Ya teniendo todo, ejecutamos la instrucción para instalar los paquetes. Esto generará una carpeta denominada "node\_modules" donde estarán alojadas las librerías declaradas en el package.json

Cuando quieras trasladar tu proyecto, puedes eliminar esta carpeta, y luego en donde la dejes, volver a ejecutar el comando para re-instalar las librerías.



# Constantes de configuracion

El archivo `.env` catalogado como dotfile, es un archivo especial donde se almacenaran todos las constantes de configuración, entre ellos:



```
.env
1 APP_PORT=3000
2 DB_HOST=127.0.0.1
3 DB_PORT=3306
4 DB_USER=root
5 DB_PASS=
6 DB_NAME=hermes_dev
```

APP\_PORT: Puerto por el cual se desplegará la API

DB\_HOST: IP o URL de donde se conectará la base de datos MySQL

DB\_PORT: Puerto de conexión al motor MySQL

DB\_USER: Nombre de usuario del motor MySQL

DB\_PASS: Clave asociada al usuario (si corresponde)

DB\_NAME: Nombre de la base de datos

# Configurando el Middleware

```
JS db.js ×
1  import mysql from 'mysql2/promise';
2
3  export default class DB {
4    constructor() {
5      this.pool = null;
6    }
7
8    setDataConnection(data){
9      this.dataConnection = data;
10     this.pool = mysql.createPool({
11       ...this.dataConnection,
12       waitForConnections: true,
13       connectionLimit: 10,
14       queueLimit: 0,
15       multipleStatements: true
16     });
17   }
18
19   async mysqlquery(query){
20     const connection = await this.pool.getConnection();
21     try {
22       const qSQL = connection.format(query);
23       const [results] = await connection.query(qSQL);
24       return {success: true, data: results};
25     } catch(error){
26       return {success: false, error: error.message};
27     } finally{
28       if(connection){
29         connection.release();
30       }
31     }
32   }
33 }
```

Un middleware es una función (o funciones) que se ejecutan entre la recepción de una solicitud y el envío de una respuesta, actuando como un puente o capa intermedia

En este caso el archivo `./middleware/db.js` será la capa intermedia que nos permitirá acceder a la base de datos MySQL y ejecutar los comandos SQL necesarios para el C.R.U.D.

# CREANDO UN SERVICIO PASO A PASO

A. E.: Crear Back-End de Aplicaciones Web con Asistencia de Inteligencia Artificial.

# Creando un servicio

```
JS tenants.services.js ×  
1  import { v4 as uuidv4 } from 'uuid';  
2  export default (db) => ({  
3    >   async findAll(){ ...  
24     },  
25     >   async findById(id){ ...  
46     },  
47     >   async create(data){ ...  
71     },  
72     >   async update(id, data){ ...  
96     },  
97     >   async delete(id){ ...  
116    }  
117  });
```

Un servicio, permite interactuar con la base de datos a través del middleware.

Cada función representa a una acción general o específica, la cual será invocada por una o varias rutas de nuestra API

La estructura de la imagen representa un ejemplo del servicio "tenants", ubicado en ./services/tenants.service.ts y que se encargará de gestionar los datos de los arrendatarios del sistema

Podrás ver que hay funciones nombradas: `findAll()` y `findById(id)` nos permitirán obtener registros; `create(data)` nos permitirá crear registros; `update(id, data)`, nos permitirá actualizar datos del registro de acuerdo al ID declarado y `delete` que permitirá eliminar el registro con el ID definido.



# Creando un servicio: función findAll()

```
async findAll(){
  const query = 'SELECT * FROM tenants';
  try {
    const results = await db.mysqlquery(query);
    const totalCount = results.data.length;
    const HTTPCode = totalCount > 0 ? 200 : 404;
    return {
      success: true,
      status: HTTPCode,
      data: results.data,
      totalCount,
      error: null};
  } catch(error){
    console.log(error);
    return {
      success: false,
      status: 500,
      data: [],
      totalCount: 0,
      error: error.message};
  }
}
```

Esta función devolverá todos los registros que se encuentren en la tabla. Para ello, construimos la instrucción SQL respectiva en la constante **query**. Dentro del bloque try { } catch() realizaremos la llamada al middleware mediante **db.mysqlquery()** pasando como parámetro, la consulta respectiva.

Si todo es exitoso hasta aquí, la consulta debería entregarnos un listado con los registros existentes. Para contarlos, usamos la constante **totalCount** y dependiendo de ello, devolvemos el código HTTP que corresponda: 404 si no hay datos; 200 si la consulta esta OK y con datos.

En el caso que todo siga su curso, construimos un JSON que servirá como respuesta; en caso de error, también habrá un JSON de respuesta, solo que el código HTTP devuelto será 500. Para mayor referencia, deberás conocer los códigos HTTP

# Creando un servicio: función findById(id)

```
async findById(id){  
  const query = `SELECT * FROM tenants WHERE id = '${id}'`;   
  console.log(query);  
  try {  
    const results = await db.mysqlquery(query);  
    const totalCount = results.data.length;  
    const HTTPCode = totalCount > 0 ? 200 : 404;  
    return {  
      success: true,  
      status: HTTPCode,  
      data: results.data,  
      totalCount,  
      error: null};  
  }catch(error){  
    return {  
      success: false,  
      status: 500,  
      data: [],  
      totalCount: 0,  
      error: error.message};  
  }  
},
```

La únicas diferencias entre esta función y la anterior son que esta devolverá cero (0) o un (1) registro nada mas.

Semánticamente es igual, solo que cambian el nombre de la función, el parámetro de entrada y el formato de la constante query.

Procedemos a los mismos controles anteriores, usando como posibles código HTTP de respuesta:

200: si la consulta es exitosa, hay coincidencia con el id y por ende, un registro

404: si la consulta es exitosa pero no hay coincidencia con el id. No hay datos que mostrar

500: si la consulta no tuvo éxito por algún error interno

# Creando un servicio: función create(data)

```
async create(data){
  const id = uuidv4();
  const query = `INSERT INTO tenants (id, name,
    contact_email, contact_phone)
  VALUES ('${id}', '${data.name}',
    '${data.contact_email}', '${data.contact_phone}')`;
  try {
    const results = await db.mysqlquery(query);
    const affectedRows = results.data.affectedRows;
    const HTTPCode = affectedRows > 0 ? 201 : 500;
    return {
      success: true,
      status: HTTPCode,
      data: {id: id},
      totalCount: affectedRows,
      error: null;
    };
  } catch(error){
    return {
      success: false,
      status: 500,
      data: [],
      totalCount: 0,
      error: error.message;
    };
  }
},
```

Esta función nos permitirá almacenar registros en la BD. Para ello, se deberá construir la instrucción respectiva en la constante **query**. Recuerda que éste es solo un ejemplo; la construcción de la query y sus características dependerá de la descripción de la tabla.

Al ejecutar una instrucción insert, la función db.mysqlquery devolverá siempre un objeto con estos tres índices:

- a) affectedRows: El número de filas afectadas por el insert.
- b) insertId: Si la tabla usa autoincrement en su clave primaria, devolverá el número auto generado.
- c) warningStatus: El numero de advertencias generadas durante la ejecución

Aquí algo importante: el HTTP Code en caso de que la inserción sea exitosa, debe ser 201 (create) a diferencia de las anteriores que usábamos 200 (OK)

# Creando un servicio: función update(id, data)

```
async update(id, data){
  let query = `UPDATE tenants SET
  name = '${data.name}',
  contact_email = '${data.contact_email}',
  contact_phone = '${data.contact_phone}'
  WHERE id = '${id}';`;
  try {
    const results = await db.mysqlquery(query);
    const affectedRows = results.data.affectedRows;
    const HTTPCode = affectedRows > 0 ? 200 : 404;
    return {
      success: true,
      status: HTTPCode,
      data: results,
      totalCount: 0,
      error: null};
  }catch(error){
    return {
      success: false,
      status: 500,
      data: [],
      totalCount: 0,
      error: error.message};
  },
}
```

Esta función nos permitirá editar registros en la BD. Es por eso que se requiere la ID que nos servirá para individualizar el registro, y la data que se va a actualizar.

Quien nos determinará si se ejecutó todo correcto, será affectedRows y según el resultado y con ello, entregar el código HTTP correspondiente.



# Creando un servicio: función delete(id)

```
async delete(id){
  let query = `DELETE FROM tenants WHERE id = ${id}`;
  try {
    const results = await db.mysqlquery(query);
    const HTTPCode = results.length > 0 ? 200 : 404;
    return {
      success: true,
      status: HTTPCode,
      data: results,
      totalCount: 0,
      error: null};
  } catch(error){
    return {
      success: false,
      status: 500,
      data: [],
      totalCount: 0,
      error: error.message};
  }
}
```

Esta función nos permitirá eliminar un registro en la BD de acuerdo al identificador asignado. Es por eso que se requiere la ID que nos servirá para individualizar el registro y evitar que se elimine otro por error.

Quien nos determinará si se ejecutó todo correcto, será `affectedRows` y según el resultado y con ello, entregar el código HTTP correspondiente. Código HTTP 200 para resultados satisfactorios, y 404 si no se encuentra el ID del registro; 500 si hay error de programación

# ENRUTADORES

A. E.: Crear Back-End de Aplicaciones Web con Asistencia de Inteligencia Artificial.

# Creando un enrutador

JS tenants.routes.js X

```
1 import express from 'express';
2 import TenantsService from '../services/tenants.services.js';
3 const router = express.Router();
4
5 export default (db) => {
6   const Tenants = TenantsService(db);
7   /** Ruta para obtener todos los registros */
8   > router.get('/', async (req, res) => { ...
15   });
16   /** Ruta para obtener un solo registro */
17   > router.get('/:id', async (req, res) => { ...
24   });
25   /** Ruta para insertar datos */
26   > router.post('/', async (req, res) => { ...
33   });
34   /** Ruta para actualizar datos (total) */
35   > router.put('/:id', async (req, res) => { ...
42   });
43   /** Ruta para eliminar datos */
44   > router.delete('/:id', async (req, res) => { ...
51   });
52   return router;
53 }
54
55
```

Los enrutadores son los que nos permitirán definir los nombres de los endpoints y los verbos que se usarán en cada uno. Generalmente las rutas que usen acciones de PUT, PATCH o DELETE, usarán el parámetro ID para limitar los registros de acción.

Cada verbo, usa dos constantes internas:

**req (Request):** Es parte del contenido que emite el front, y contiene el cuerpo de las peticiones o requerimientos.

**res (Response):** Es parte de la respuesta que se le entregará al solicitante una vez procesado el requerimiento.

# Creando un enrutador: obtener todos los registros

```
router.get('/', async (req, res) => {  
  const qData = await Tenants.findAll(); //Invocamos al método del servicio  
  let data = qData.data; //Obtenemos los datos  
  let status = qData.status; //Obtenemos el código HTTP para respuesta  
  let totalCount = qData.totalCount; //Obtenemos el número de registros  
  let error = qData.error; //Obtenemos el mensaje en caso de error  
  res.status(status).json({ data, totalCount, error });  
});
```

Al usar el verbo GET, estamos haciendo solicitud de los datos al servidor. En este ejemplo, invocaremos al método `findAll` creado previamente en el servicio, para obtener todos los datos y con los resultados crearemos un JSON que usaremos como respuesta utilizando `res` como base e indicando el estado (`status`)



# Creando un enrutador: obtener un registro específico

```
/** Ruta para obtener un solo registro */  
router.get('/:id', async (req, res) => {  
  const qData = await Tenants.findById(req.params.id);  
  let data = qData.data;  
  let status = qData.status;  
  let totalCount = qData.totalCount;  
  let error = qData.error;  
  res.status(status).json({ data, totalCount, error });  
});
```

Al igual que el anterior, usamos GET como verbo, y podremos notar solo 2 diferencias: el uso de un identificador en el endpoint (/:id), y la invocación del método findById, pasado como parámetro de entrada, el identificador del endpoint.

Siempre que queramos obtener datos pasado por la URL del endpoint, debemos usar req.params, que es una colección de parámetros capturadas por Express

# Creando un enrutador: Crear un registro

```
/** Ruta para insertar datos */  
router.post('/', async (req, res) => {  
  const qData = await Tenants.create(req.body);  
  let data = qData.data;  
  let status = qData.status;  
  let totalCount = qData.totalCount;  
  let error = qData.error;  
  res.status(status).json({ data, totalCount, error });  
});
```

Para la creación de registros, siempre debemos usar el verbo POST. Este verbo normalmente no requiere parámetros por URL, por lo que la información por parte del cliente, debe ser enviada en formato JSON mediante el cuerpo del requerimiento (de ahí que usemos req.body)

El resto, es igual que el resto de los endpoints que hemos creado hasta aquí.

# Creando un enrutador: Editar un registro

```
/** Ruta para actualizar datos (total) */  
router.put('/:id', async (req, res) => {  
  const qData = await Tenants.update(req.params.id, req.body);  
  let data = qData.data;  
  let status = qData.status;  
  let totalCount = qData.totalCount;  
  let error = qData.error;  
  res.status(status).json({ data, totalCount, error });  
});
```

Los verbos PUT y PATCH nos permiten editar registros. Inicialmente PUT fue concebido para actualizar todos los elementos de un registro, mientras que PATCH para cambiar solo algunos. En este caso usaremos PUT. También es importante saber que se deben usar tanto los parámetros por URL (el id en /:id), y el cuerpo del JSON de petición donde vendrán los cambios (req.body)

# Creando un enrutador: Eliminar un registro

```
/** Ruta para eliminar datos */  
router.delete('/:id', async (req, res) => {  
  const qData = await Tenants.delete(req.params.id);  
  let data = qData.data;  
  let status = qData.status;  
  let totalCount = qData.totalCount;  
  let error = qData.error;  
  res.status(status).json({ data, totalCount, error });  
});
```

De igual manera como lo requiere el método para edición, el método para eliminar registros requerirá un identificador recepcionado por la URL. Internamente usamos el método DELETE quien se encargará de procesar el requerimiento. Si puede notar, esta vez no requerimos datos por el cuerpo del requerimiento, a diferencia de PUT o POST



# NUCLEO

A. E.: Crear Back-End de Aplicaciones Web con Asistencia de Inteligencia Artificial.

# index.js: El núcleo de nuestra aplicación

```
JS index.js ×
1  import dotenv from "dotenv";
2  import express from "express";
3  import http from "http";
4  import cors from 'cors';
5
6  import DB from "../middleware/db.js";
7
8  import tenantRouterFactory from "../routes/tenants.routes.js";
9
10 dotenv.config();
11
12 const initServer = () => {
13   const app = express();
14   const server = http.createServer(app);
15   const db = new DB();
16   const dConnection = {
17     host: process.env.DB_HOST,
18     user: process.env.DB_USER,
19     password: process.env.DB_PASSWORD,
20     database: process.env.DB_NAME,
21     port: process.env.DB_PORT
22   };
23   db.setDataConnection(dConnection);
24
25   app.use(express.json());
26   app.use(express.urlencoded({ extended: true }));
27   app.use(cors({
28     origin: '*',
29     methods: ['GET', 'POST', 'PATCH', 'OPTIONS'],
30     allowHeaders: ['Content-Type', 'Authorization', 'X-Requested-With', 'Accept', 'Origin'],
31     credentials: true
32   }));
33   /** creamos una constante con el enrutador de los propietarios */
34   const tenantRouter = tenantRouterFactory(db);
35   /* Configuramos una ruta para los propietarios */
36   app.use("/tenants", tenantRouter);
37
38   /* Configuramos la ruta por defecto (/) */
39   app.get("/", (req, res) => {
40     res.status(200).json({ message: "API REST funcionando correctamente" });
41   });
42
43   server.listen(process.env.APP_PORT, () => {
44     console.log("Servidor funcionando en el puerto " + process.env.APP_PORT);
45   });
46 }
47 initServer();
48
```

El archivo index.js (puedes ponerle el nombre que tu quieras mientras lo identifiques como main en el archivo package.json), es quien orquesta toda la aplicación.

- ✓ Captura los datos de conexión a la BD desde el .env
- ✓ Gestiona la conexión con la base de datos
- ✓ Convierte los datos recibidos por cada requerimiento a un formato legible
- ✓ Determina los nombres de las rutas que usaremos
- ✓ Configura CORS para permitir conexiones remotas

Deglosemos el documento de ejemplo para usar sus referencias:

# index.js: Deglose - librerías

```
JS index.js ×
1  import dotenv from "dotenv";
2  import express from "express";
3  import http from "http";
4  import cors from 'cors';
5
6  import DB from "../middleware/db.js";
7
8  import tenantRouterFactory from "../routes/tenants.routes.js";
9
10 dotenv.config();
```

Al inicio, debemos importar las librerías que nuestro archivo necesita. Aquí como ejemplo veras dotenv para procesar el archivo .env donde esta la configuración, express como framework, http como procesador de requerimientos a través de este protocolo y cors para procesar peticiones externas.

En adelante, ya pasa a ser parte de nuestro código personalizado:

- DB corresponde al middleware
- tenantRouterFactory corresponde al archivo donde configuramos el enrutador.
  - Aquí, tendremos que repetir esta línea por cada nueva referencia a otro enrutador
- Dotenv.config() nos permite obtener los valores ingresados en nuestro archivo .env

# index.js: Deglose - librerías

```
12  ✓ const initServer = () => {
13      const app      = express();
14      const server    = http.createServer(app);
15      const db        = new DB();
16  ✓  const dConnection = {
17          host: process.env.DB_HOST,
18          user: process.env.DB_USER,
19          password: process.env.DB_PASSWORD,
20          database: process.env.DB_DATABASE,
21          port: process.env.DB_PORT
22      }
23      db.setDataConnection(dConnection);
24
25      app.use(bodyParser.json());
26      app.use(express.json());
27      app.use(express.urlencoded({ extended: true }));
28
29      const tenantRouter = tenantRouterFactory(db);
30      /* Configuramos una ruta para los propietarios */
31      app.use("/tenants", tenantRouter);
32
33      /* Configuramos la ruta por defecto (/) */
34  ✓  app.get("/", (req, res) => {
35          res.status(200).json({ message: "API REST funcionando correctamente" });
36      });
37
38  ✓  server.listen(process.env.APP_PORT, () => {
39          console.log("Servidor funcionando en el puerto " + process.env.APP_PORT);
40      });
41  }
42  initServer();
```

La función `initServer` será la encargada de comenzar todas las operaciones.

Puedes notar que entre las líneas 13 y 15 tenemos referencias a `express`, `http` y nuestro middleware `DB`. Este último, al ser una clase, necesitamos crear un json para traspasar los datos de conexión al middleware (creado en la constante `dConnection` entre las filas 16 y 22) específicamente en la fila 23.

Luego cargamos las configuraciones a nuestra app de `bodyParser` y `express` (líneas 25 a 27)

Las filas 29 y 31 deberás repetirla por cada enrutador que hayas importado, de forma que asuma el objeto `db` creado desde la clase `DB` y pueda procesar las consultas SQL.

Para terminar, entre las líneas 34 y 36 podrás notar la configuración del endpoint inicial (/); entre las filas 38 y 40, la configuración de escucha del servidor con su despliegue en la consola del mensaje de inicio; por ultimo pero no por ello menos importante, la línea 42 invoca a la función `initServer` para su ejecución

# DESPLEGANDO LA APLICACION

A. E.: Crear Back-End de Aplicaciones Web con Asistencia de Inteligencia Artificial.



# Ejecución del servicio API

```
TERMINAL  PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  PUERTOS

🍏 > ~/Documents/Develop/API node index.js
[dotenv@17.2.3] injecting env (6) from .env -- tip: ⚙ suppress all logs with { quiet: true }
Servidor funcionando en el puerto 3000
```

Después de hacer la implementación de todos los endpoints, es hora de probar el funcionamiento del servicio, utilizando en nuestra terminal, desde el punto en que tengamos el proyecto, el comando

```
node index.js
```

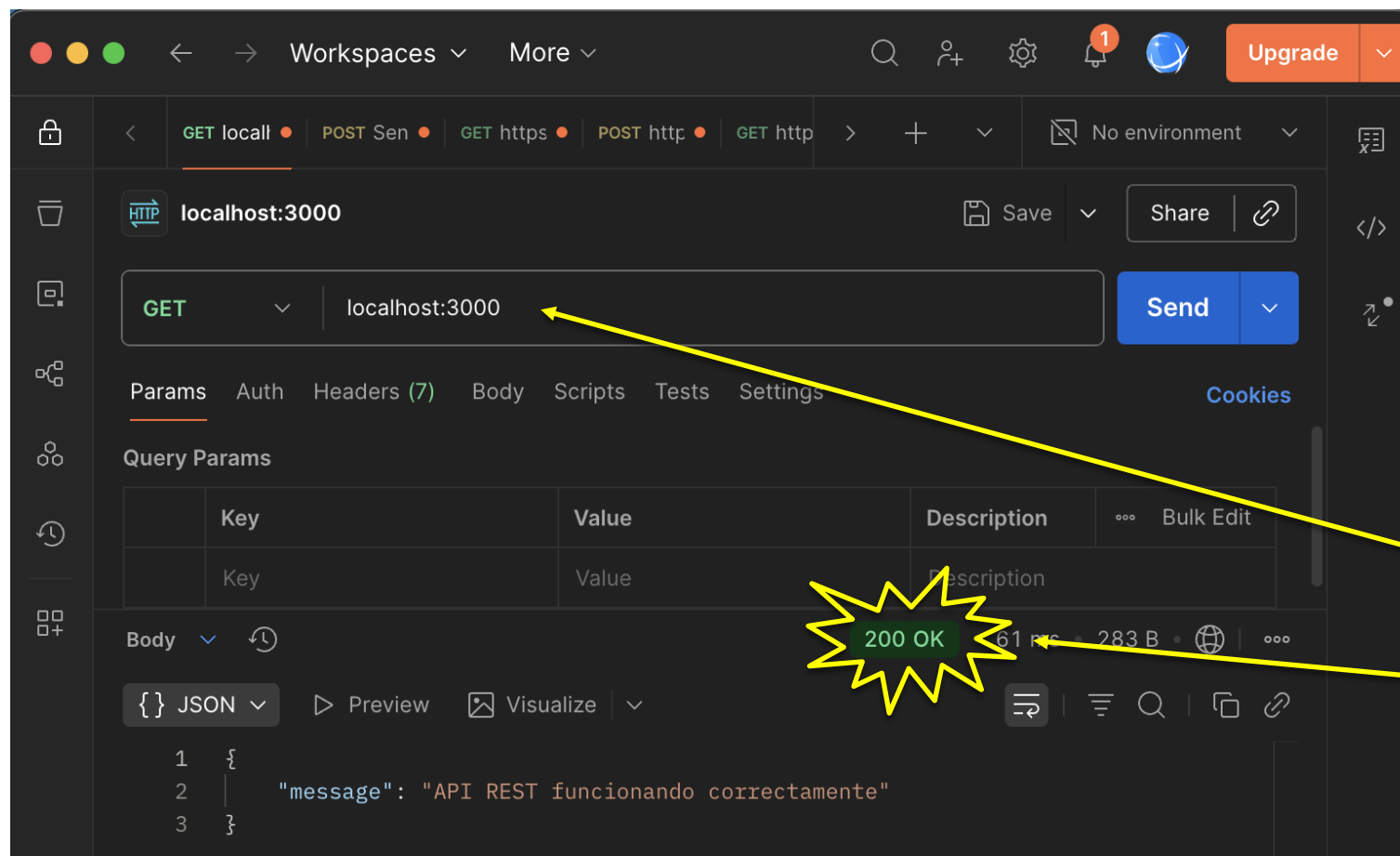
Si todo esta bien, deberíamos tener el mensaje de lanzamiento de la API, indicando que el servicio se encuentra activo, y el puerto de ejecución.

# POSTMAN: Software predilecto para pruebas



Postman es un software open-source, multiplataforma, cuyo principal objetivo es la ejecución de pruebas sobre las API Rest. Ofrece la posibilidad de lanzar peticiones a URL de API utilizando diferentes verbos, tipos de datos, Completación de formularios y cualquier otro requerimiento para pruebas, devolviéndonos información como códigos HTTP obtenidos por la ejecución, respuestas por parte del servidor, etc. Puedes descargarlo directamente de su web <https://www.postman.com/>

# POSTMAN: pruebas de operación



Ya con Postman abierto y el servicio corriendo, solo nos basta hacer las pruebas de operación.

Seleccionamos GET como verbo de prueba y digitamos la dirección en la cual esta corriendo nuestro servicio. Como está en local, y en el puerto 3000, entonces escribimos

<http://localhost:3000>

Como verás, en la parte inferior tendremos el resultado, y junto con ello el código HTTP

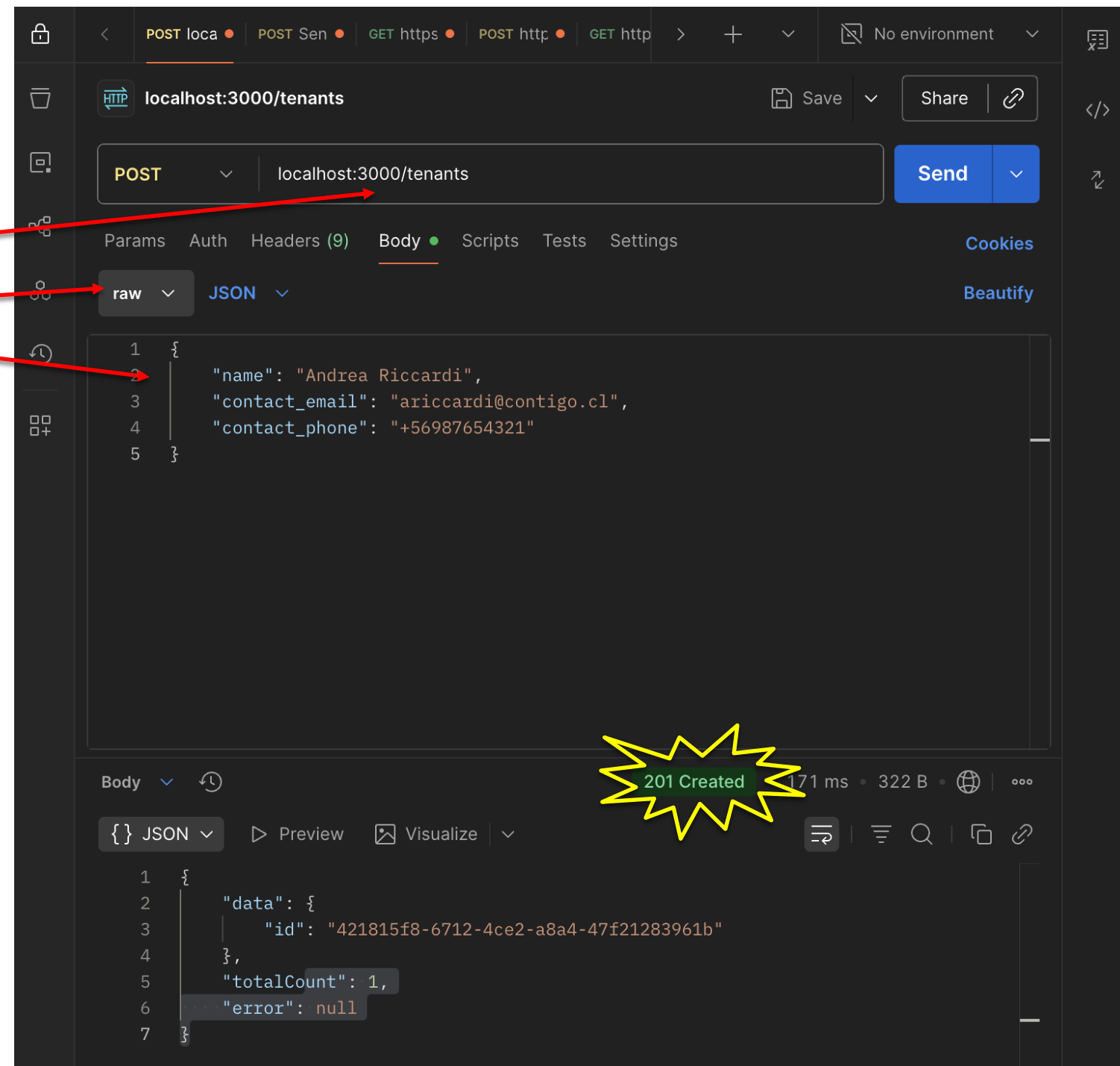
# POSTMAN: probando POST (éxito)

Para insertar datos, usaremos POST como verbo HTTP. Usaremos el ejemplo del endpoint tenants.

Los datos, se deben enviar por el cuerpo de requerimiento (body). Debes seleccionar el formato (raw) y luego, crear un objeto JSON con los campos que el endpoint requiere para su construcción

Si tu ejecución es exitosa, el endpoint de enviará el JSON de respuesta, y comunicará el código HTTP (recuerda que habíamos programado el 201 para este caso)

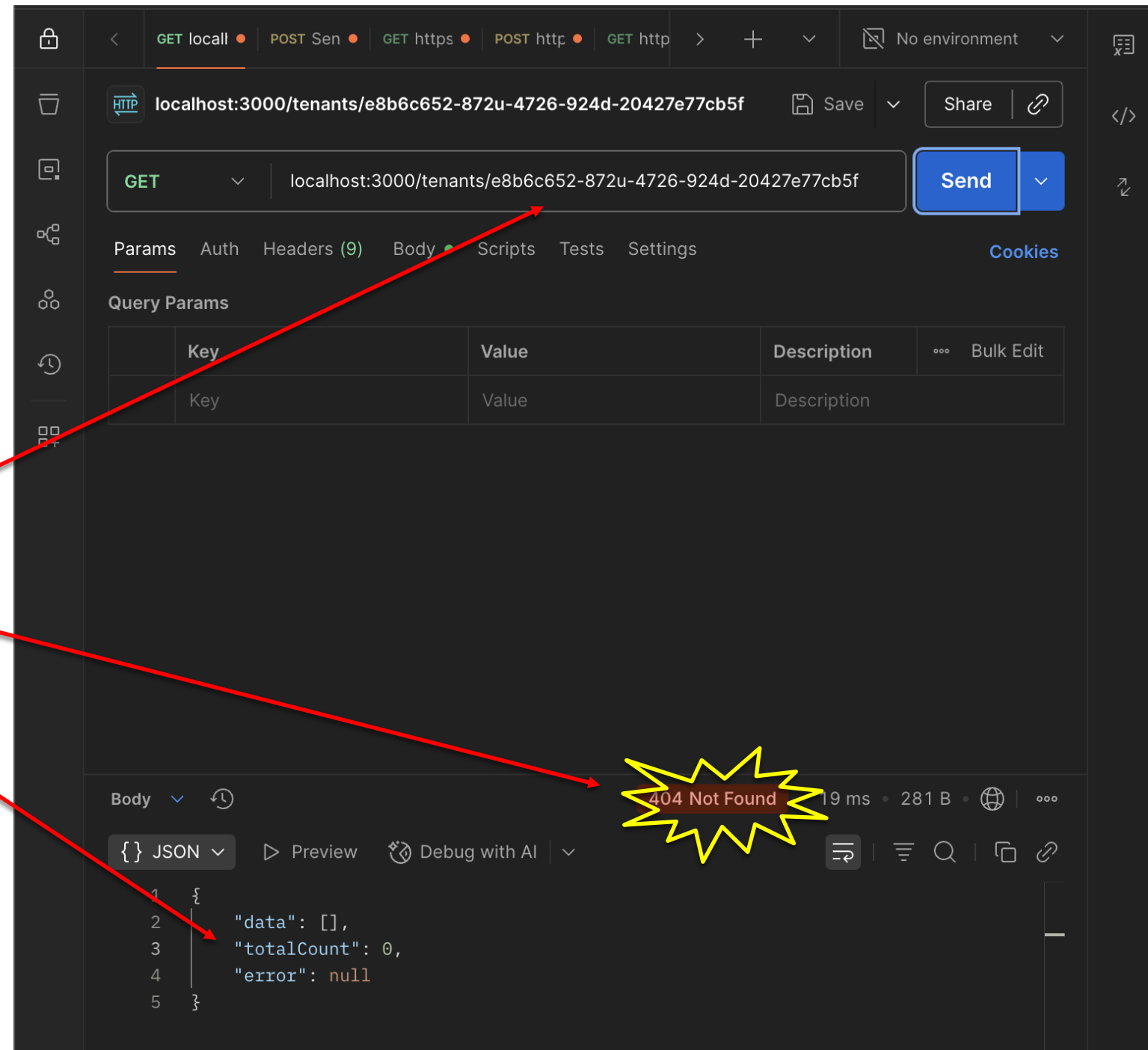
**CUIDADO:** Es altamente probable que, si no le pasas cuerpo, de igual forma te añada datos, pero con los valores undefined. Tu trabajo es poder añadir validaciones antes de hacer cualquier inserción...



# POSTMAN: probando GET (error)

El verbo GET nos permite obtener datos. Recordemos que hemos creado dos endpoints en tenants para obtener datos: el primero, sin el ID, que devolverá todos los registros, y un segundo, que tendremos que pasarle un ID para obtener algún dato

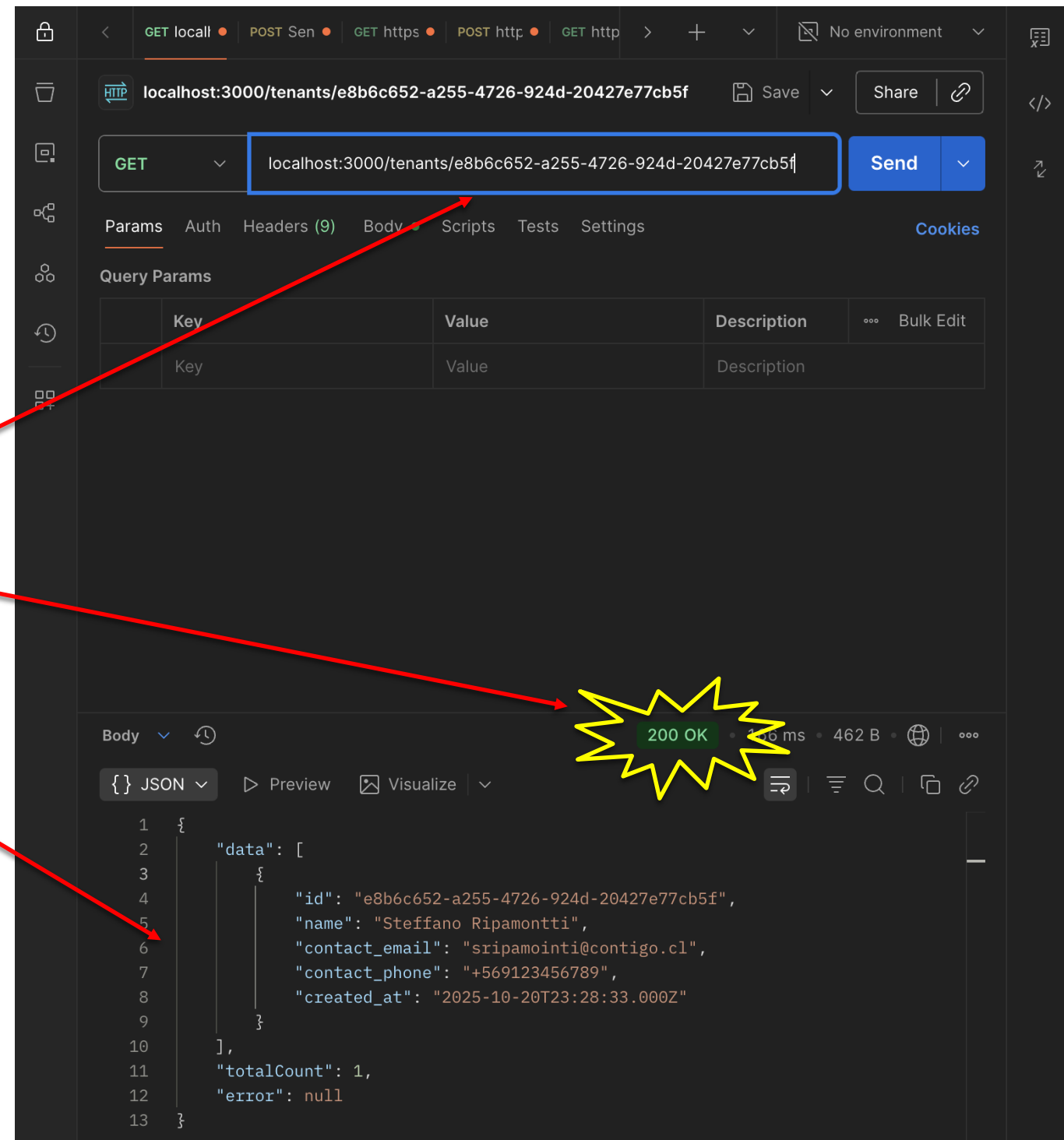
En este ejemplo, vemos como hemos usado GET a tenants, pasando un identificador incorrecto, devolviendo como respuesta, el código HTTP correspondiente y el JSON reflejando el resultado





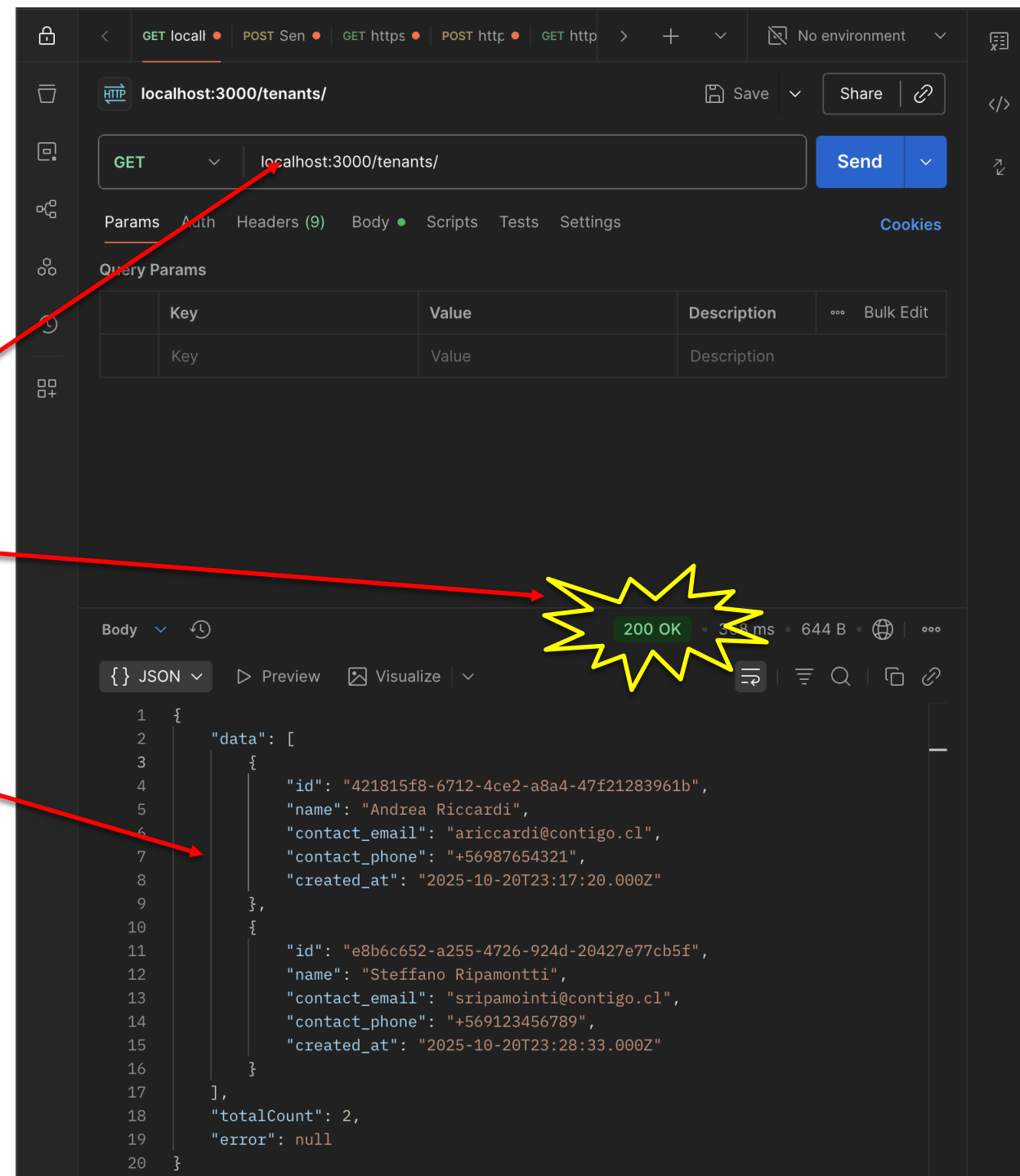
# POSTMAN: probando GET (éxito)

Ahora, por el contrario, si pasamos un ID correcto, podemos ver como cambian el código HTTP de respuesta, y el JSON con el cual tenemos detalles del procesamiento de datos.



# POSTMAN: probando GET (éxito)

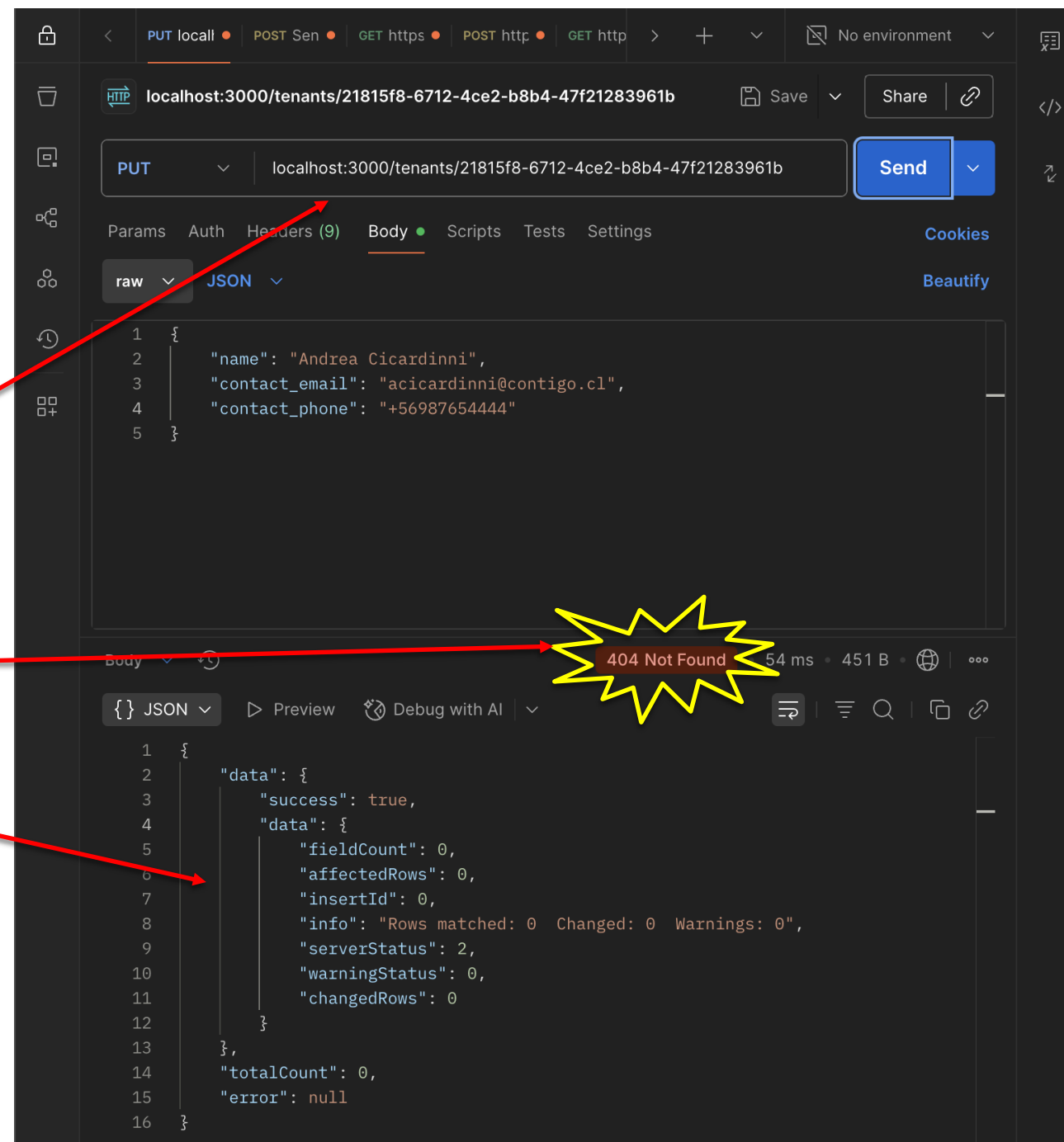
Finalmente, si accedemos directo al endpoint, sin pasar ningún ID, nos devolvería el juego completo de registros que hay en la base de datos



# POSTMAN: probando PUT (error)

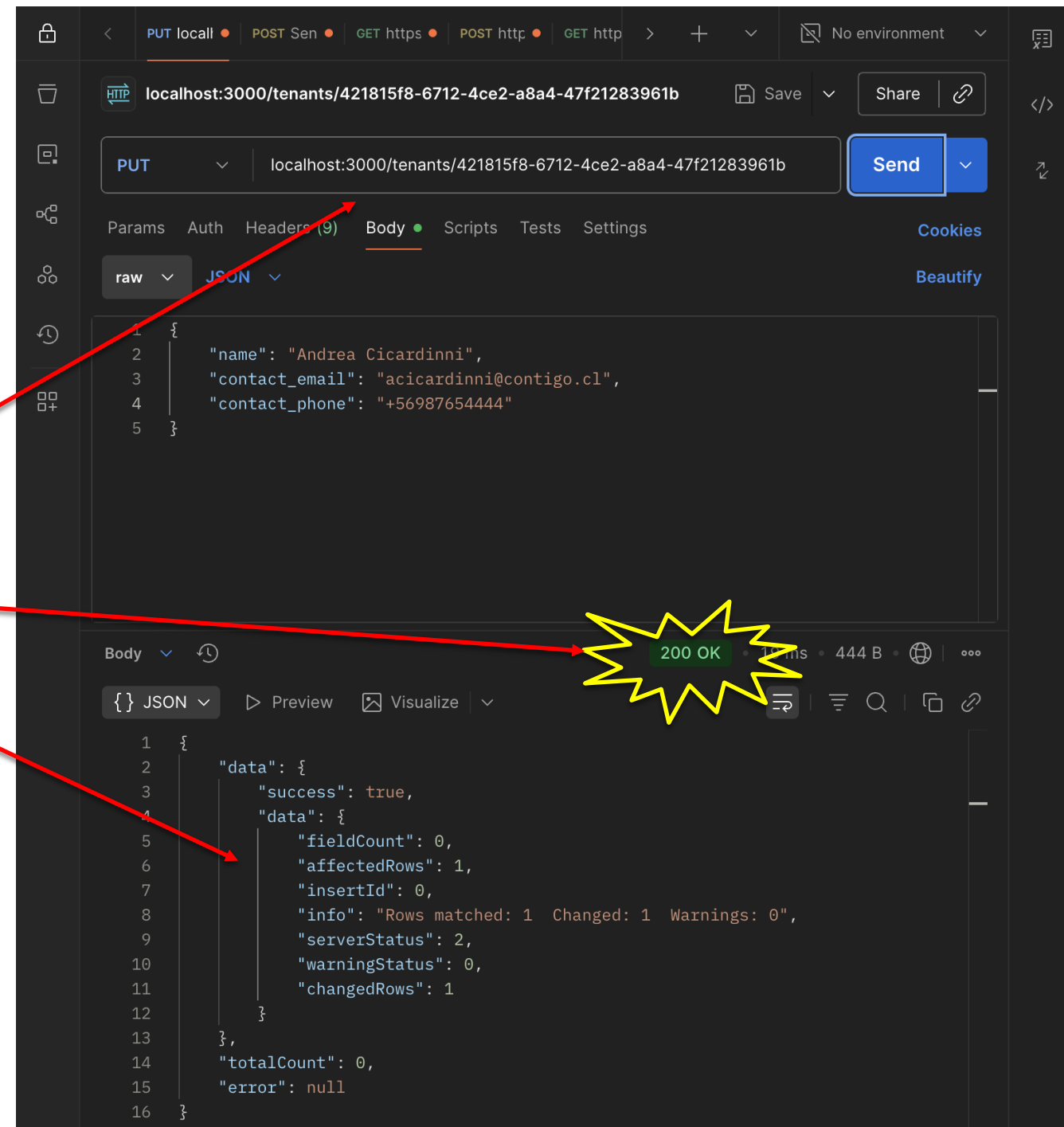
El verbo PUT nos permite hacer cambios en los datos de los registros. Para ello, y según hemos visto, en el endpoint es requisito pasar un ID válido.

En este ejemplo, intentaremos editar un registro con un ID inexistente y nos da el código 404 dado que el ID no existe



# POSTMAN: probando PUT (éxito)

Aquí podemos comprobar que efectivamente el ID si corresponde a un registro valido, por lo cual, el cambio de datos del registro fue satisfactorio, dado que el código HTTP de respuesta fue un 200



# Consideraciones

El proyecto que hemos generado es solo una muestra de como podemos implementar un servicio API Rest con NodeJS, implementando los principales verbos HTTP.

**¡ESTE PROYECTO NO ES SEGURO!**

No implementes en producción un proyecto hasta esta etapa que hemos visto, dado que tenemos muchos problemas de seguridad (que abordaremos más adelante).

Es tu misión ahora, finalizar el resto de endpoints que se requerirán para el proyecto, de acuerdo a la estructura que tenga la base de datos.





SOY TÉCNICO!

GRACIAS!



**5** Acreditado  
años  
En Gestión Institucional y  
Docencia de Pregrado  
hasta el 4 de Noviembre de 2026



SOY TÉCNICO!



**5** **Acreditado**  
**años**  
En Gestión Institucional y  
Docencia de Pregrado  
hasta el 4 de Noviembre de 2026