Reporte: Actividad 4

“Diseño de Backend con reglas de negocio, control de estados con endpoint de búsqueda”

Alumna: Jocelyn Flores Gutiérrez

Maestro: Edgar Ivan Patricio Aizpuru

Materia: Desarrollo Full Stack

Fecha: 14/02/2026

● Índice

Extender el Backend actual agregando un endpoint avanzado que permita:

- *Búsqueda por nombre*
- *Filtro por rango de precio*
- *Paginación real usando LIMIT y OFFSET*
- *Construcción segura de consultas SQL dinámicas*

Repositorio Github:
https://github.com/JoFloresGutierrez06/Actividad-4---Full-Stack

Documentación de Postman:
https://jofloresgutierrez-602355.postman.co/workspace/1f545f8f-3fde-45fa-88af-4fbc9538319c

● Requerimientos

---

1. Crear nuevo endpoint:

GET /productos/search

Debe aceptar los siguientes query params opcionales:

- ?nombre=lap
- ?minPrecio=100
- ?maxPrecio=1000
- ?page=1
- ?limit=5

2. Comportamiento esperado

El endpoint debe:

- Filtrar por nombre usando ILIKE
- Filtrar por rango de precio (minPrecio y maxPrecio)
- Ordenar por id DESC
- Aplicar paginación real con LIMIT y OFFSET
- Regresar respuesta estructurada:

Ejemplo esperado:

```
{
  "data": [...],
  "page": 1,
  "limit": 5,
  "total": 12
}
```

3. Reglas técnicas obligatorias

✔ Usar parámetros seguros $1, $2, $3
✔ No concatenar strings manualmente
✔ Validar que page y limit sean números
✔ Manejar caso cuando no haya resultados
✔ La lógica SQL debe estar en el repository, no en el controller

• Desarrollo

- **¿Cómo construiste la query dinámica?**
  En el controlador usé desestructuración para recibir los parámetros, asignando un valor default a page y limit si el usuario no indicaba nada. Luego se validó que estos últimos tuvieran un tipo de dato válido numérico y en general se comprobaba cada parámetro para asegurar que no se quedara sin un valor (o que se volviera nulo si no se había indicado), para luego pasar los parámetros a la función de buscar.

  En la función de buscar se evaluó cada uno de los casos de los parámetros otorgados (es algo ineficiente pero no se me ocurrió otra manera de hacerlo) y además de guardar los resultados de los querys, también se hizo un query para contar los elementos totales sin paginación y se devolvían ambos valores en un return de json.

  De vuelta al controlador, se evaluaba si con el método de buscar se obtuvieron elementos o no y finalmente se mostraban al usuario en un json que tenía los datos y los parámetros de page, limit y el contador de elementos.

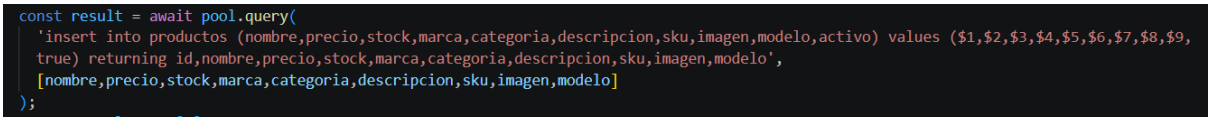
  Ejemplo:

```
async buscar({nombre, minPrecio, maxPrecio, page, limit}) {
  /* const result = await pool.query(···
  const offset = (page - 1) * limit;

  if (nombre && minPrecio && maxPrecio) {
    const result = await pool.query(
      'select id,nombre,precio,stock,marca,categoria,descripcion,sku,imagen,modelo,activo from productos where nombre ilike $1 and precio
      between $2 and $3 order by id desc limit $4 offset $5', [`%${nombre}%`,minPrecio, maxPrecio, limit, offset]
    )
    const contar = await pool.query(
      'select count(*) from productos where nombre ilike $1 and precio between $2 and $3', [`%${nombre}%`,minPrecio, maxPrecio]
    )
    const total = Number(contar.rows[0].count)
    return {resultados: result.rows, total}
  }
  else if (nombre && minPrecio) {
    const result = await pool.query(
      'select id,nombre,precio,stock,marca,categoria,descripcion,sku,imagen,modelo,activo from productos where nombre ilike $1 and precio >=
```

```
async function search(req,res) {
  /* const nombre = req.query.nombre;
  const resultados = await repo.buscar(nombre); */

  const { nombre, minPrecio, maxPrecio, page = 1, limit = 5} = req.query; // Le dejo limit 5 default para que no traiga tantos elementos

  // Validar page
  const pageNumber = Number(page)
  if(Number.isNaN(pageNumber) || pageNumber <= 0) {console.log("Entró al condicional de page. Page = "+ pageNumber)
    return res.status(400).json({error: 'Page inválida'})
  } // Validar limit
  const limitNumber = Number(limit)
  if(Number.isNaN(pageNumber) || limitNumber <= 0) { console.log("Entró al condicional de limit. Limit = "+limitNumber)
    return res.status(400).json({error: 'Limit inválido'})
  }
  const parametros = {
    nombre: nombre ?? null,                          // Revisa si hay un elemento, si no lo hay se vuelve null
    minPrecio: minPrecio ? Number(minPrecio) : null,  // Asegurarse que sea un numero
    maxPrecio: maxPrecio ? Number(maxPrecio) : null,  // Asegurarse que sea un numero
    page: page ? Number(page) : null,                 // Asegurarse que sea un numero
    limit: limit ? Number(limit) : null               // Asegurarse que sea un numero
  }

  // Buscar productos con filtros y paginación
  const resultados = await repo.buscar(parametros);

  if (!resultados.resultados || resultados.resultados.length === 0) {
    return res.status(404).json({error: 'Ningún producto encontrado'})
  }

  return res.json({
    data: resultados.resultados,
    page: parametros.page,
    limit: parametros.limit,
    total: resultados.total
  });
```

- **¿Cómo evitaste SQL Injection?**
  Se evitó principalmente con el método que se vió en clase de no insertar las variables directamente en las instrucciones de SQL, sino de forma indirecta utilizando comillas simples para toda la instrucción e insertando indirectamente las variables en un arreglo y referenciándolas.

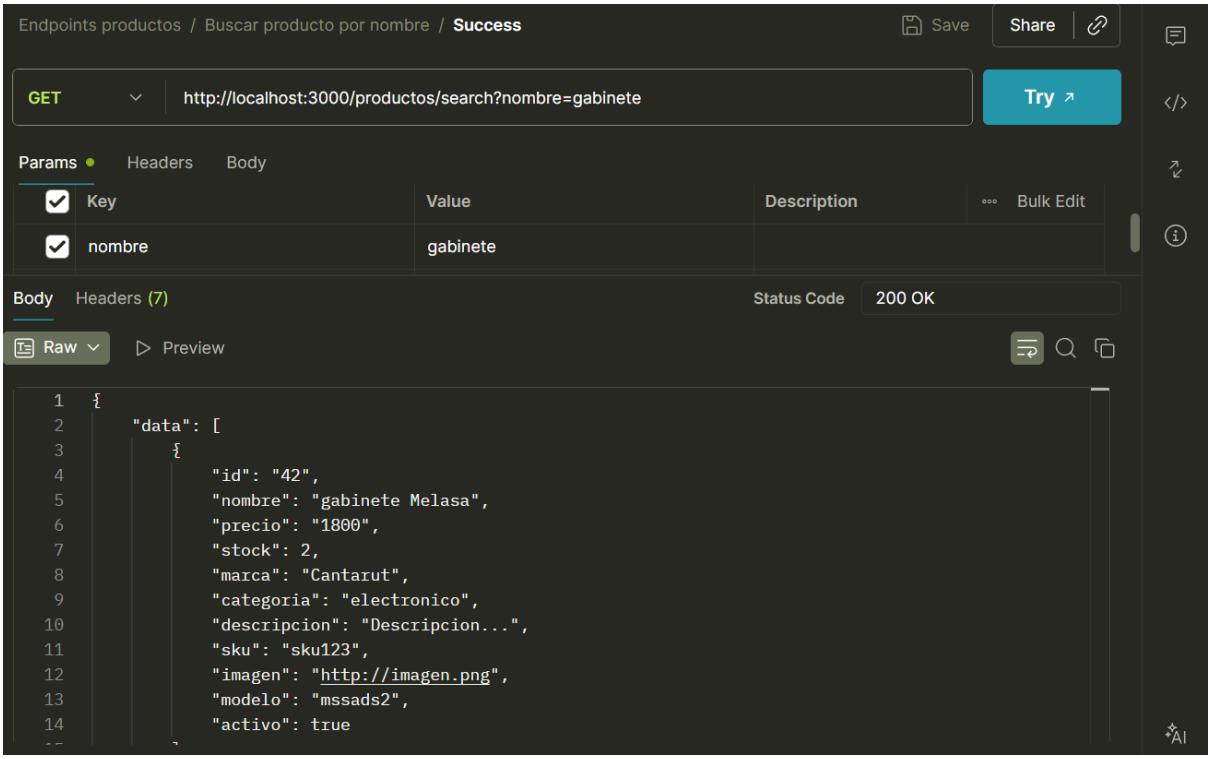
  Ejemplo:



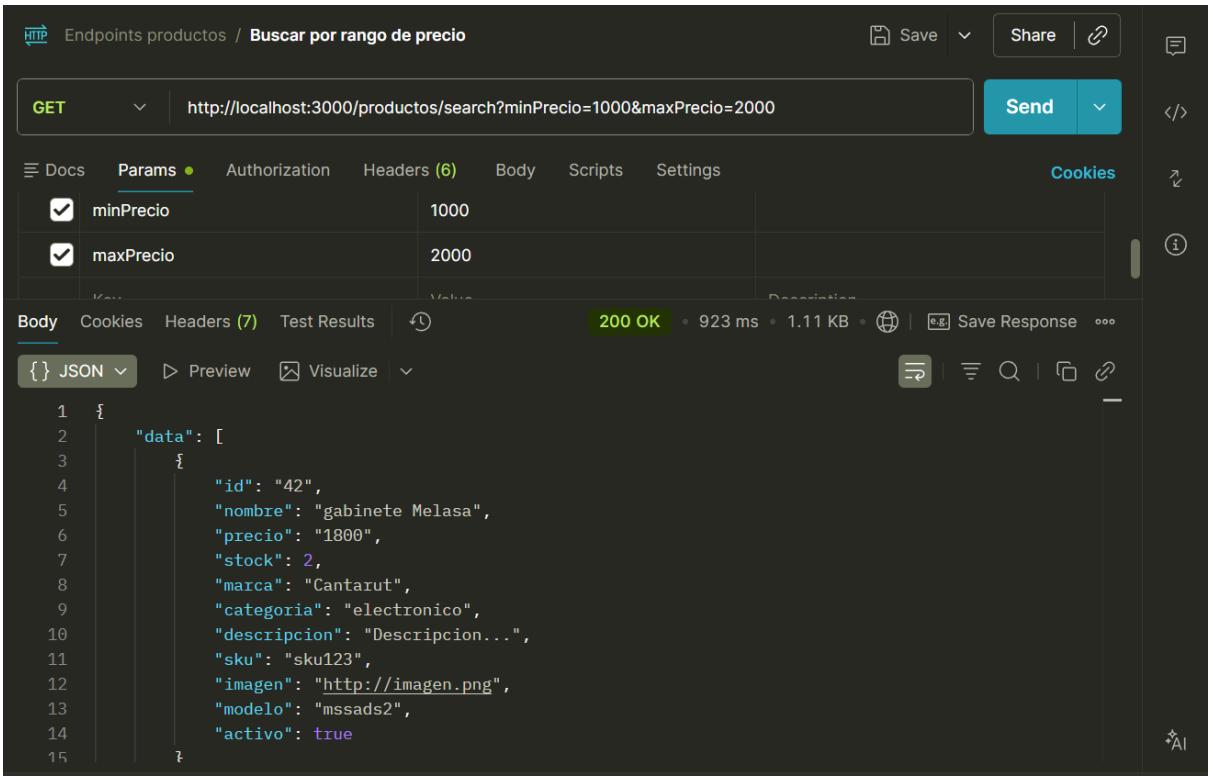
```
const result = await pool.query(
  'insert into productos (nombre,precio,stock,marca,categoria,descripcion,sku,imagen,modelo,activo) values ($1,$2,$3,$4,$5,$6,$7,$8,$9,
  true) returning id,nombre,precio,stock,marca,categoria,descripcion,sku,imagen,modelo',
  [nombre,precio,stock,marca,categoria,descripcion,sku,imagen,modelo]
);
```

• Postman

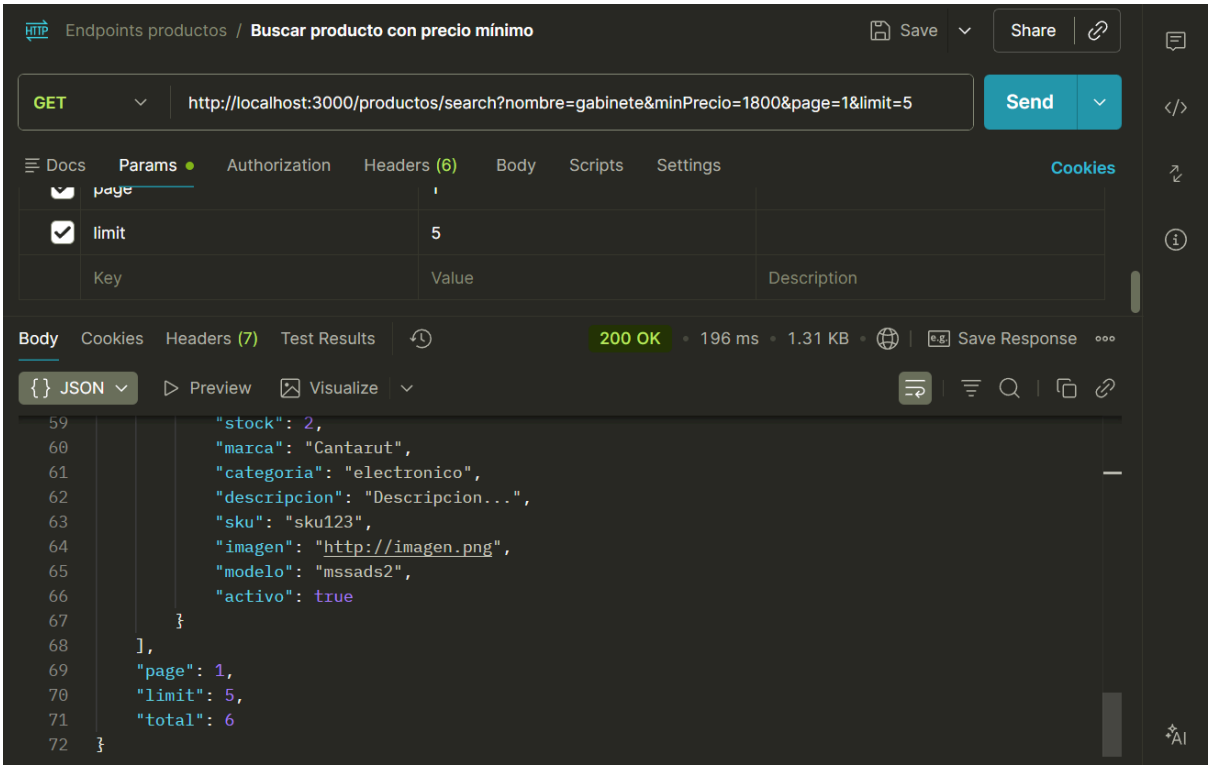
- Solo búsqueda por nombre:



- Solo rango de precio:



- Combinación nombre + rango:



- Uso de paginación: