

Guía de estudio 5 - Diseño avanzado de una API REST



¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿En qué consiste esta guía?

La siguiente guía de estudio tiene como objetivo recordar y repasar los contenidos que hemos visto en clase.

Dentro de los que se encuentran:

- Especificar el límite de registros que se obtiene al consultar una ruta GET de una API REST
- Ordenar los registros de una API en base a query strings.
- Numerar los registros que se devuelven en una consulta GET de una API REST.
- Filtrar los registros que se devuelven en base a los parámetros recibidos en la query strings.

¡Vamos con todo!



Tabla de contenidos

Limitar los registros de una consulta	3
¡Manos a la obra! - GET /personal con limit	6
Ordenamiento de registros en una consulta	7
¡Manos a la obra! - GET /personal con Order by	9
Paginación de registros en una consulta	9
¡Manos a la obra! - GET /personal con page	11
Filtrado de registros en una consulta	13
¡Manos a la obra! - GET /personal con Order by	14
Filtros de registros con consulta parametrizada	15
¡Manos a la obra! - Obtener Personal por Filtros	16
HATEOAS	16
¡Manos a la obra! - Implementa HATEOAS	19
La ruta por defecto	20
¡Manos a la obra! - Servidor personal	21
Preguntas	21



¡Comencemos!

Limitar los registros de una consulta

¿Por qué limitar la cantidad de registros de una consulta?

Una API que devuelve 10000 registros de una base de datos probablemente demorará mucho más que una API que devuelve 10 registros.

Dicho de otra forma, el tiempo que se demora en obtener una respuesta a una consulta de un cliente dependerá de la cantidad de información que procesará la API y el tamaño de la respuesta.

Por este y otros motivos, las páginas web que manejan grandes cantidades de datos como e-commerce o aplicaciones de inventarios, recurren a limitar el consumo de los recursos, buscando un mejor rendimiento e incluso una mejor presentación y experiencia de usuario.

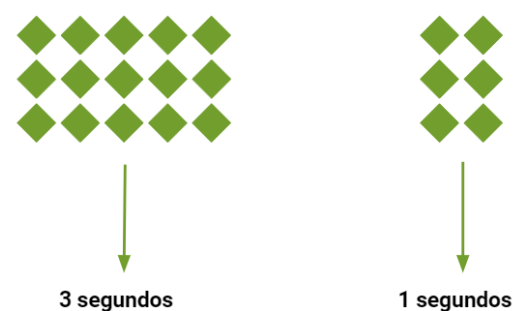


Imagen 1. Diferencia al limitar registros
Fuente: Desafío Latam

En esta unidad aprenderemos a limitar el tamaño de las consultas e incluso a paginar resultados.

Para empezar necesitaremos descargar un ZIP de la plataforma que contiene el código base para nuestro proyecto.

Una vez descargado crearemos una API REST que sirva para obtener los datos de los medicamentos y el personal de una farmacia, para esto debes ejecutar el archivo script.sql o simplemente copiar y pegar el código en la terminal psql.

```

scripts.sql
1 CREATE DATABASE farmacia;
2 \c farmacia;
3
4 CREATE TABLE medicamentos (id SERIAL, nombre VARCHAR(50) NOT NULL, precio INT NOT NULL, stock INT NOT NULL CHECK (stock >= 0) );
5
6 INSERT INTO medicamentos values
7 (DEFAULT, 'Paracetamol', 3500, 25 ),
8 (DEFAULT, 'Ibuprofeno', 6900, 10 ),
9 (DEFAULT, 'Orfidal', 10900, 30 ),
10 (DEFAULT, 'Zolpidem', 5000, 12 ),
11 (DEFAULT, 'Mentix', 35900, 22 ),
12 (DEFAULT, 'Diclofenaco', 1900, 50 ),
13 (DEFAULT, 'Frenadol', 4900, 41 ),
14 (DEFAULT, 'Piretanyl', 1290, 7 ),
15 (DEFAULT, 'Aspirina', 4500, 35 ),
16 (DEFAULT, 'Ventolin', 22900, 5 );
17
18 CREATE TABLE personal (id SERIAL, nombre VARCHAR(50) NOT NULL, rol VARCHAR(25) NOT NULL, salario INT NOT NULL );
19
20 INSERT INTO personal values
21 (DEFAULT, 'Jane Margolis', 'administrador', 5000),
22 (DEFAULT, 'Skyler White', 'cajero', 3500),
23 (DEFAULT, 'Ignacio Vargas', 'administrador', 2200),
24 (DEFAULT, 'Walter White', 'farmaceutico', 7000),
25 (DEFAULT, 'Jesse Pinkman', 'farmaceutico', 6500),
26 (DEFAULT, 'Gustavo Fring', 'gerente', 10000),
27 (DEFAULT, 'Saul Goodman', 'abogado', 4000),
28 (DEFAULT, 'Hank Schrader', 'seguridad', 1500),
29 (DEFAULT, 'Mike Ehrmantraut', 'seguridad', 1750);
30
31 SELECT * FROM medicamentos;
32 SELECT * FROM personal;

```

Imagen 2. Script SQL para iniciar

Fuente: Desafío Latam

Para los ejercicios guiados utilizaremos la tabla **medicamentos**:

```

farmacia=# SELECT * FROM medicamentos;
 id |  nombre  | precio | stock 
----+-----+-----+-----
  1 | Paracetamol |   3500 |    25 
  2 | Ibuprofeno |   6900 |    10 
  3 | Orfidal    |  10900 |    30 
  4 | Zolpidem   |   5000 |    12 
  5 | Mentix     |  35900 |    22 
  6 | Diclofenaco |   1900 |    50 
  7 | Frenadol   |   4900 |    41 
  8 | Piretanyl  |   1290 |     7 
  9 | Aspirina   |   4500 |    35 
 10 | Ventolin   |  22900 |     5 
(10 filas)

```

Imagen 3. Tabla medicamentos

Fuente: Desafío Latam

Para los ejercicios propuestos utilizaremos la tabla **personal**:

```
farmacia=# SELECT * FROM personal;
```

id	nombre	rol	salario
1	Jane Margolis	administrador	5000
2	Skyler White	cajero	3500
3	Ignacio Vargas	administrador	2200
4	Walter White	farmaceutico	7000
5	Jesse Pinkman	farmaceutico	6500
6	Gustavo Fring	gerente	10000
7	Saul Goodman	abogado	4000
8	Hank Schrader	seguridad	1500
9	Mike Ehrmantraut	seguridad	1750

(9 filas)

Imagen 4. Tabla personal
Fuente: Desafío Latam

Para permitirle a aplicaciones clientes la posibilidad de decidir cuántos medicamentos recibirá en una consulta a nuestra API REST, crearemos una función que realice una consulta SQL para obtener todos los medicamentos agregando la cláusula **LIMIT**

```
// consultas.js
// ...

// Agregamos la función obtenerMedicamentos
const obtenerMedicamentos = async ({ limits = 10 }) => {
  let consulta = "SELECT * FROM medicamentos LIMIT $1"
  const { rows: medicamentos } = await pool.query(consulta,
[limits])
  return medicamentos
}

// Exportamos la función
module.exports = {obtenerMedicamentos}
```

La cláusula limit puede recibir la cantidad de resultados a limitar, pero si se especifica all puede devolver todos los resultados, de esta forma si quisiéramos que nuestra endpoint fuera capaz de devolver todos los registros podríamos utilizar un limits = 'all' como parámetro por defecto.

El siguiente paso será llamar a esta función desde el archivo index en la ruta **GET /medicamentos**

```
// index.js
// importamos la función
const { obtenerMedicamentos } = require('./consultas')
// Utilizamos el query recién creado

app.get('/medicamentos', async (req, res) => {
  const queryStrings = req.query
  const medicamentos = await obtenerMedicamentos(queryStrings)
  res.json(medicamentos)
})
```

Ahora probemos la ruta y función creada realizando una consulta con Thunder Client que especifique un límite de 3 en la query strings de la URL.

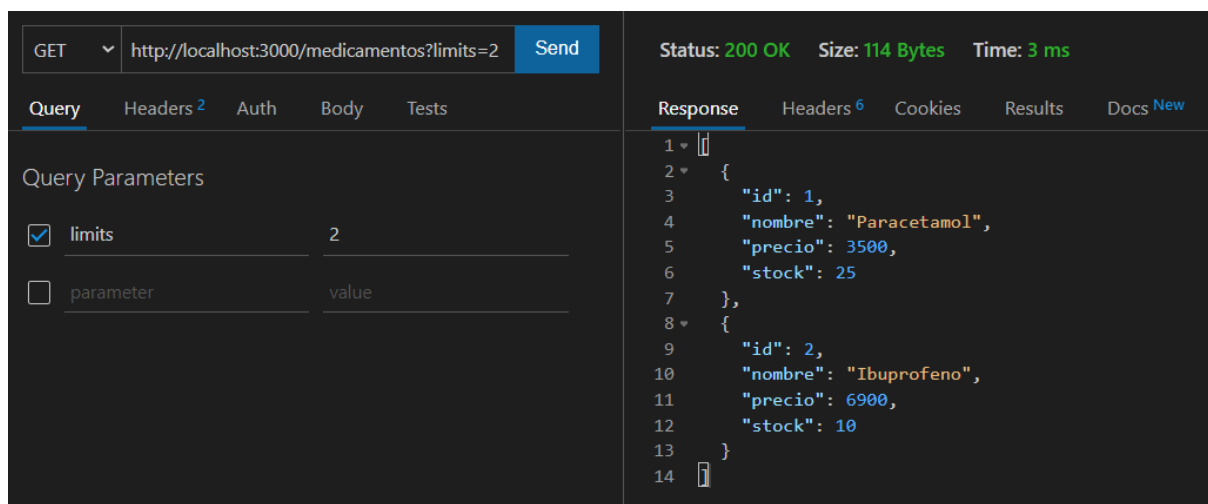


Imagen 5. Probando la ruta GET /medicamentos con limits
Fuente: Desafío Latam



¡Manos a la obra! - GET /personal con limit

Crea una función para obtener el personal de la farmacia que sea utilizada en una ruta: **GET /personal**

Debes incluir la posibilidad de limitar la cantidad de registros con el parámetro limits.

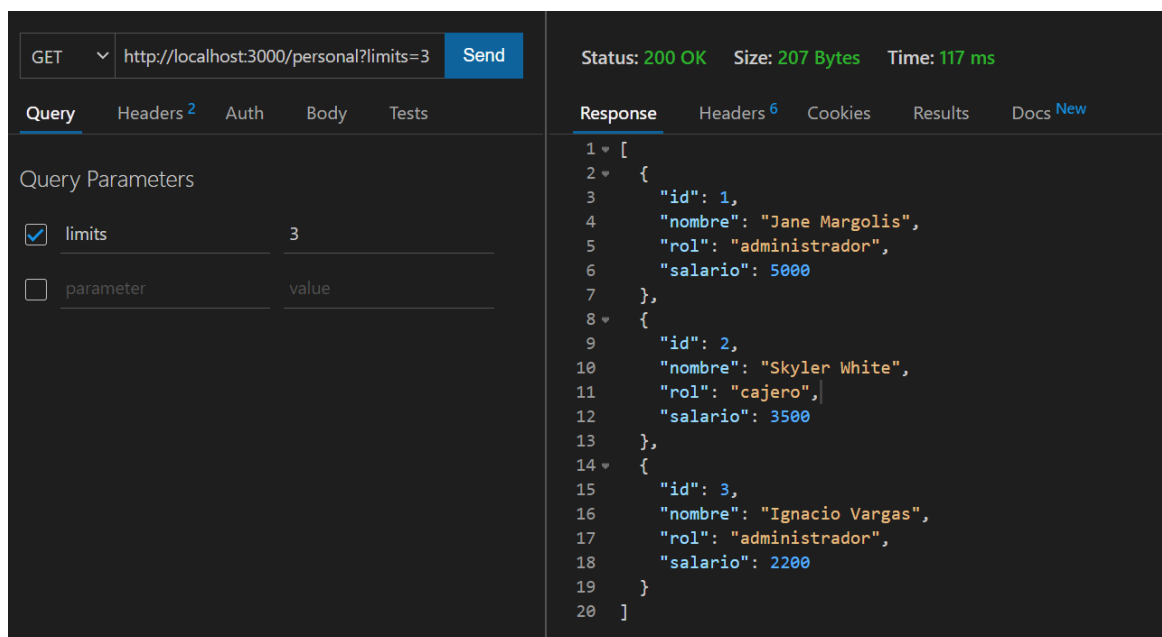


Imagen 6. Probando la ruta GET /personal con limits
Fuente: Desafío Latam

Ordenamiento de registros en una consulta

De la misma manera que recibimos un parámetro **limits** por query strings, podemos recibir otros parámetros que podemos utilizar para ordenar nuestros registros.

Existen varias maneras de especificar en el valor de un parámetro cuál es el campo y el orden deseado, una de estas consiste en separar ambas declaraciones por un guion bajo (_).

Por ejemplo:

- stock_ASC
- stock_DESC
- precio_ASC
- precio_DESC
- nombre_ASC
- nombre_DESC

La dificultad es que para lograr esto tenemos que referirnos a columnas de nuestra base de datos dentro de la consulta, este tipo de consultas recibe el nombre de consultas dinámicas o (dynamic queries). Para evitar problemas de sql injection en este tipo de consultas tendremos que instalar un paquete nuevo llamado **pg-format**

En el terminal, dentro de la carpeta del proyecto, ejecutamos:

```
npm install pg-format
```

Luego al principio del archivo consultas.js incorporaremos la biblioteca en nuestro proyecto con

```
// consultas.js
const format = require('pg-format');
```

En la función **obtenerMedicamentos** dentro del mismo archivo **consulta.js** agregaremos otro parámetro para indicar la columna y dirección con la que ordenaremos los resultados.

```
// consultas.js
// ...

const obtenerMedicamentos = async ({ limits = 10, order_by =
"id_ASC" }) => {

  const [campo, direccion] = order_by.split("_")
  const formattedQuery = format('SELECT * FROM medicamentos
order by %s %s LIMIT %s', campo, direccion, limits);

  const { rows: medicamentos } = await
pool.query(formattedQuery)
  return medicamentos
}
```

Como el nombre del campo vendrá así: stock_ASC tenemos que separarlo por el guion abajo y eso nos dará el campo stock y la dirección asc.

Para hacer el query sin abrir espacios a sql injection, utilizaremos format, con %s indicaremos los campos y valores, luego ejecutaremos el query de la misma forma que lo hemos hecho previamente.

Con la función modificada consultemos nuevamente la ruta **GET /medicamentos** agregando un parámetro **order_by** que tenga como valor: **precio_DESC**

GET http://localhost:3000/medicamentos?limits=2&order_by=precio_DESC Send

Status: 200 OK Size: 109 Bytes Time: 4 ms

Query Parameters

Parameter	Value
limits	2
order_by	precio_DESC
parameter	value

Response

```
1 * [
2 *   {
3 *     "id": 5,
4 *     "nombre": "Mentix",
5 *     "precio": 35900,
6 *     "stock": 22
7 *   },
8 *   {
9 *     "id": 10,
10 *    "nombre": "Ventolin",
11 *    "precio": 22900,
12 *    "stock": 5
13 *   }
14 * ]
```

Imagen 7. Probando la ruta GET /medicamentos con limits y order_by
Fuente: Desafío Latam



¡Manos a la obra! - GET /personal con Order by

Crea una función para obtener el personal de la farmacia que sea utilizada en una ruta: **GET /personal**

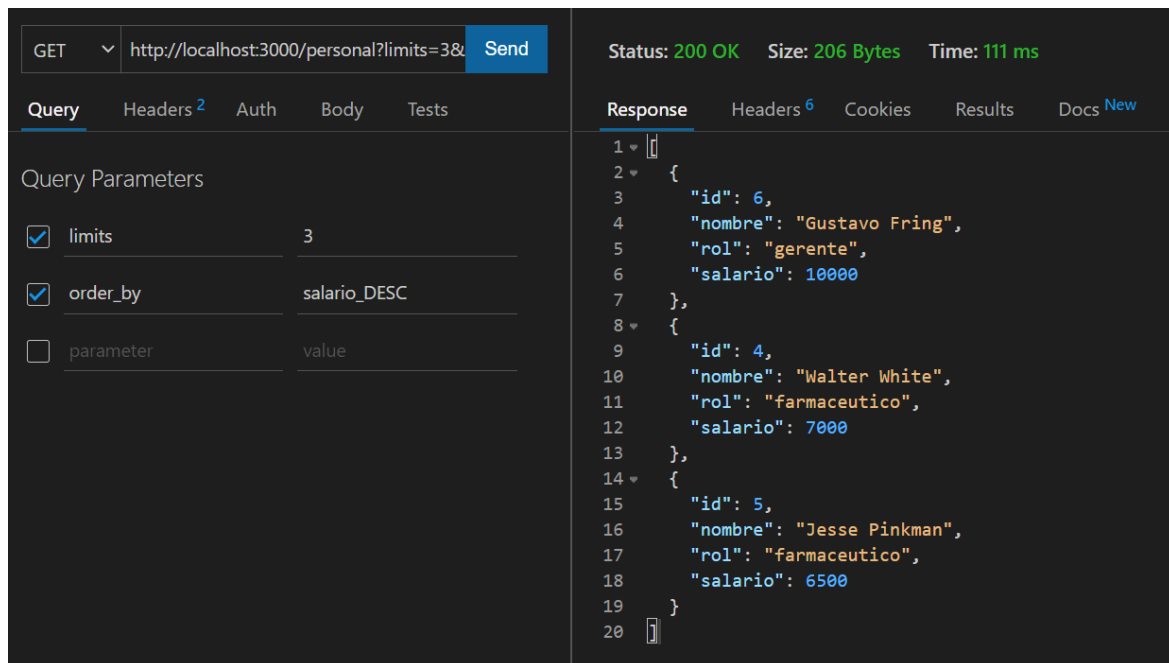


Imagen 8. Probando la ruta GET /personal con limits y order_by

Fuente: Desafío Latam

Paginación de registros en una consulta

En conjunto con la limitación de medicamentos, podemos generar una paginación de datos que distribuya en cada página la misma cantidad de registros y mejore la navegación de estos.

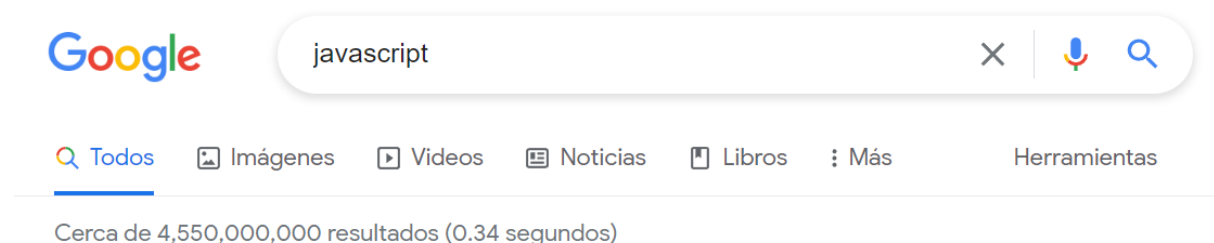




Imagen 9. Ejemplo de paginación con Google
Fuente: Desafío Latam

Para esto necesitaremos agregar un condicional en la función **obtenerMedicamentos** que agregue un OFFSET a la consulta SQL

```
const obtenerMedicamentos = async ({ limits = 10, order_by = "id_ASC",  
page = 0 }) => {  
  
  const [campo, direccion] = order_by.split("_")  
  const offset = page * limits  
  
  const formattedQuery = format('SELECT * FROM medicamentos order by  
%s %s LIMIT %s OFFSET %s', campo, direccion, limits, offset);  
  
  pool.query(formattedQuery);  
  
  const { rows: medicamentos } = await pool.query(formattedQuery)  
  return medicamentos  
}
```

El **OFFSET** nos permite omitir una cantidad de filas en una consulta SQL.

Supongamos que tenemos 23 registros en una tabla y queremos obtener todas las filas omitiendo las primeras 9:

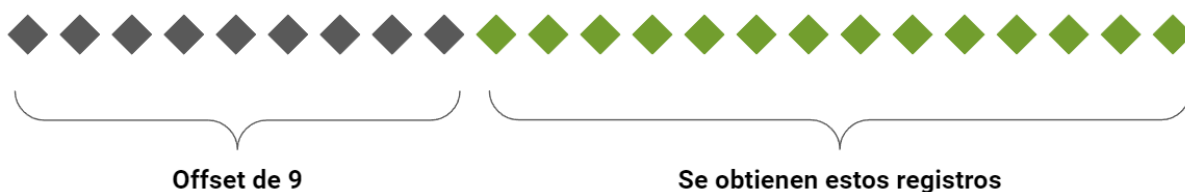


Imagen 10. Diagrama de ejemplo sobre el Offset
Fuente: Desafío Latam

Utilizando la misma idea, si tenemos páginas de 10 resultados, en la página 0, el offset será 0, en la página 1, el offset será 10, en la página 2 el offset será 20 y así. Por lo que podemos calcular el offset como $page * limits$.

Si quisiéramos que la primera página fuera la página 1 y no la página 0, tendríamos que modificar el código para partir por defecto en la página 1 y calcular el offset como $(page - 1)$

* limits

```
const obtenerMedicamentos = async ({ limits = 10, order_by = "id_ASC",  
page = 1 }) => {  
  
  const [campo, direccion] = order_by.split("_")  
  const offset = (page - 1) * limits  
  
  const formattedQuery = format('SELECT * FROM medicamentos order by  
%s %s LIMIT %s OFFSET %s', campo, direccion, limits, offset);  
  
  pool.query(formattedQuery);  
  
  const { rows: medicamentos } = await pool.query(formattedQuery)  
  return medicamentos  
}
```

Consultemos nuevamente la ruta **GET /medicamentos** agregando un parámetro **page** que tenga como valor: **2**

The screenshot shows a web browser with the URL `http://localhost:3000/medicamentos?limits=2&order_by=precio_DESC&page=3`. The query parameters are listed as follows:

Query Parameters	Value
limits	2
order_by	precio_DESC
page	3
parameter	value

The response is a JSON array of 3 medication objects:

```
[  
  {  
    "id": 3,  
    "nombre": "Orfidal",  
    "precio": 10900,  
    "stock": 30  
  },  
  {  
    "id": 9,  
    "nombre": "Aspirina",  
    "precio": 4500,  
    "stock": 35  
  },  
  {  
    "id": 7,  
    "nombre": "Frenadol",  
    "precio": 4900,  
    "stock": 41  
  }  
]
```

Imagen 11. Probando la ruta GET /medicamentos con limits, order_by y page
Fuente: Desafío Latam



¡Manos a la obra! - GET /personal con page

Permite la paginación de los registros del personal usando el parámetro **page** obtenido en la query string de la ruta **GET /personal**

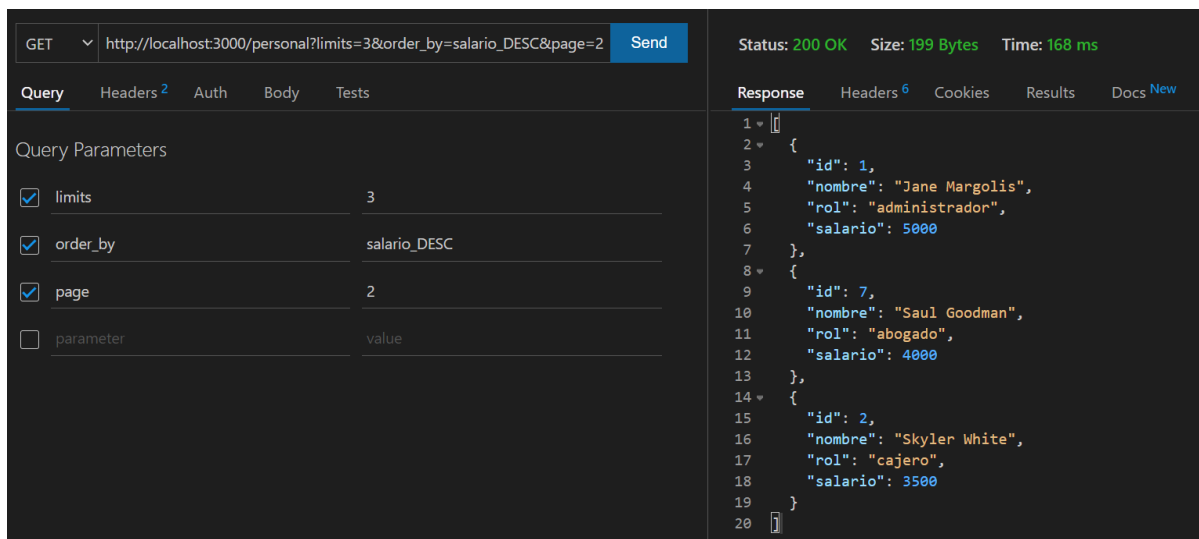


Imagen 12. Probando la ruta GET /personal con limits, order_by y page
Fuente: Desafío Latam

Filtrado de registros en una consulta

Para filtrar los medicamentos podemos evaluar la existencia de varios otros parámetros como:

- stock_min
- precio_min
- stock_max
- precio_max

Cada uno de ellos, en caso de existir, deberán agregar en la consulta SQL una condición según sea el caso.

Creemos una nueva función llamada **obtenerMedicamentosPorFiltros** que reciba las query strings de una consulta y prepare una consulta que incluya la palabra reservada WHERE y considere la inclusión de la palabra AND por cada uno de los filtros recibidos:

```
const obtenerMedicamentosPorFiltros = async ({ stock_min, precio_max }) => {
  let filtros = []
  if (precio_max) filtros.push(`precio <= ${precio_max}`)
  if (stock_min) filtros.push(`stock >= ${stock_min}`)

  let consulta = "SELECT * FROM medicamentos"
  if (filtros.length > 0) {
    filtros = filtros.join(" AND ")
    consulta += ` WHERE ${filtros}`
  }
  const { rows: medicamentos } = await pool.query(consulta)
  return medicamentos
}
```

Se utilizan los arreglos para ir agregando todas las condiciones y finalmente unirlos e incluir el WHERE en la consulta.

La función anterior la vamos a utilizar en una nueva ruta dedicada a consumir los registros de los medicamentos con filtros:

```
app.get('/medicamentos/filtros', async (req, res) => {
  const queryStrings = req.query
  const medicamentos = await obtenerMedicamentosPorFiltros(queryStrings)
  res.json(medicamentos)
})
```

Consultemos la nueva ruta **GET /medicamentos/filtros** pasando como query strings los parámetros **precio_max** y **stock_min** con los valores 4000 y 25 respectivamente.

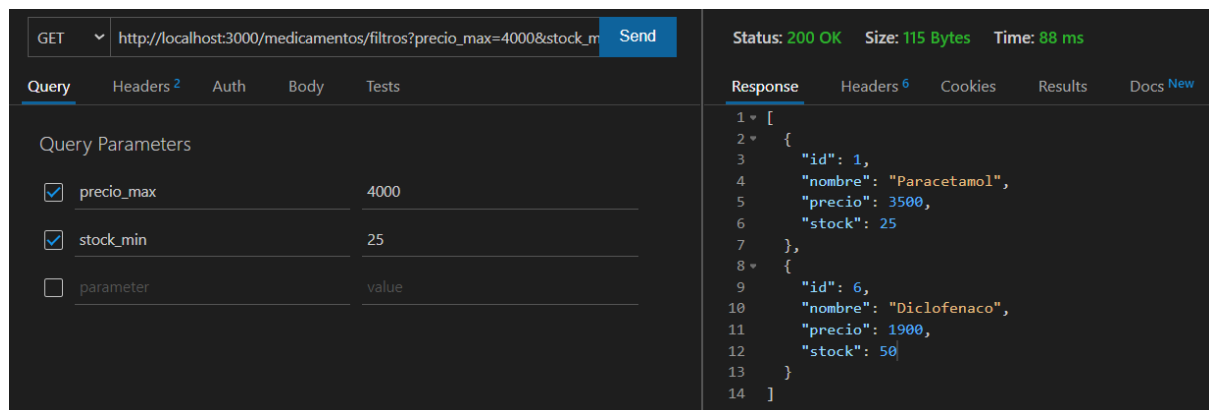


Imagen 13. Probando la ruta GET /medicamentos/filtros con filtros
Fuente: Desafío Latam



¡Manos a la obra! - GET /personal con Order by

Crea una ruta **GET /personal/filtros** que obtenga por query strings los parámetros: **salario_max**, **salario_min** y **rol**.

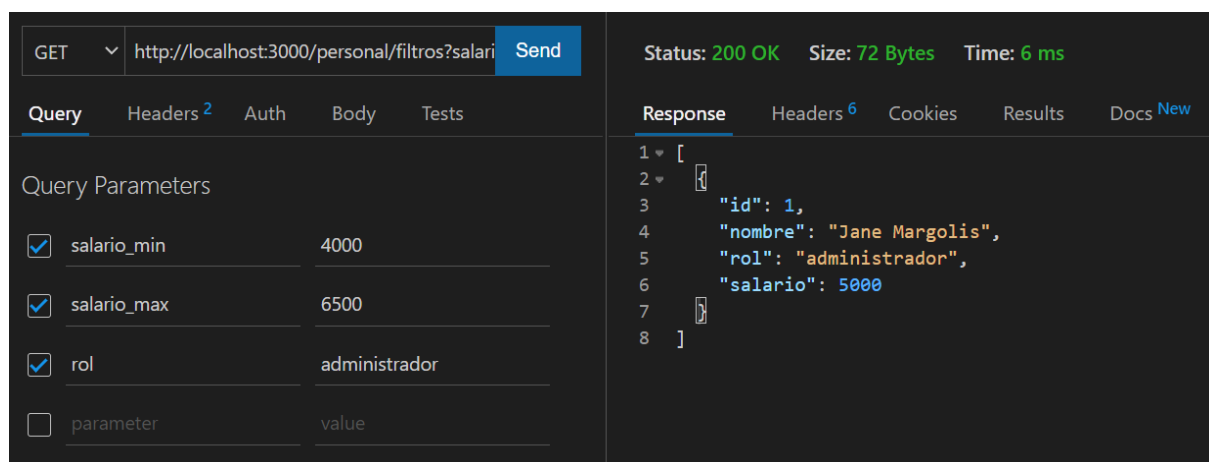


Imagen 13. Probando la ruta GET /personal/filtros con filtros
Fuente: Desafío Latam

Filtros de registros con consulta parametrizada

Para evitar la inyección SQL en la obtención de medicamentos por filtros, reescribamos la función **obtenerMedicamentosPorFiltros** de la siguiente manera:

```
const obtenerMedicamentosPorFiltros = async ({ precio_max, stock_min }) => {
  let filtros = []
  const values = []

  const agregarFiltro = (campo, comparador, valor) => {
    values.push(valor)
    const { length } = filtros
    filtros.push(`${campo} ${comparador} ${length + 1}`)
  }

  if (precio_max) agregarFiltro('precio', '<=', precio_max)
  if (stock_min) agregarFiltro('stock', '>=', stock_min)

  let consulta = "SELECT * FROM medicamentos"

  if (filtros.length > 0) {
    filtros = filtros.join(" AND ")
    consulta += ` WHERE ${filtros}`
  }

  const { rows: medicamentos } = await pool.query(consulta, values)
  return medicamentos
}
```

Dentro de la función estamos creando otra función llamada **agregarFiltro** que nos servirá para agregar un filtro en el arreglo **filtros** y el valor correspondiente al parámetro, siendo ejecutada solo en caso de que se validen la existencia de los parámetros en la query string.

Si probamos nuevamente la obtención de medicamentos por filtros veremos que sigue funcionando sin problemas:

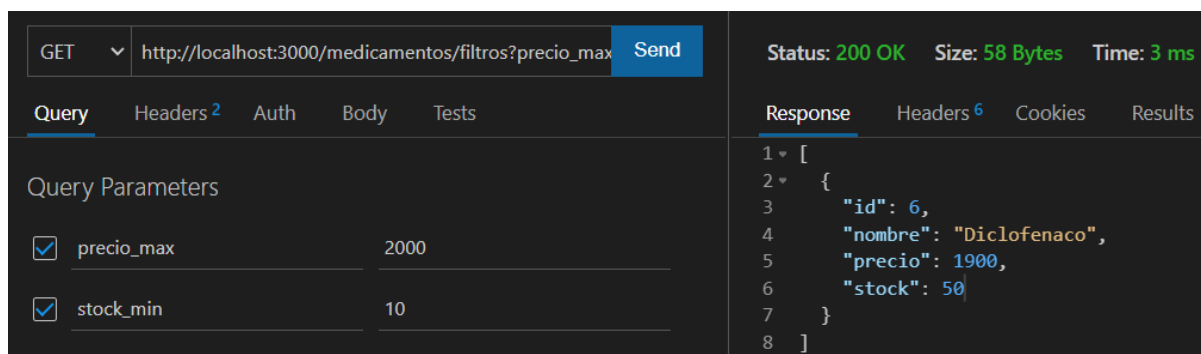


Imagen 14. Consulta parametrizada para la obtención de medicamentos con filtros
Fuente: Desafío Latam



¡Manos a la obra! - Obtener Personal por Filtros

Reescribe la función **obtenerPersonalPorFiltros** para que realice la consulta SQL de forma parametrizada

HATEOAS

Por sus siglas en inglés Hypermedia As The Engine Of Application State (Hipermedia como motor del estado de la aplicación), es un modelo de datos que podemos adoptar en nuestras API REST basado en recursos y sub recursos relacionados por enlaces.

Un ejemplo conocido de este modelo se puede apreciar con la [Poke API](#):

```

4 {
5   "count": 1118,
6   "next": "https://pokeapi.co/api/v2/pokemon/?offset=20&limit=20",
7   "previous": null,
8   "results": [
9     {
10      "name": "bulbasaur",
11      "url": "https://pokeapi.co/api/v2/pokemon/1/"
12    },
13    {
14      "name": "ivysaur",
15      "url": "https://pokeapi.co/api/v2/pokemon/2/"
16    },
17    {
18      "name": "venusaur",
19      "url": "https://pokeapi.co/api/v2/pokemon/3/"
20    },
21    {
22      "name": "charmander",
23      "url": "https://pokeapi.co/api/v2/pokemon/4/"
  
```

Imagen 14. Ejemplo de HATEOAS de la pokeapi.

Fuente: [PokeApi](#)

Como puedes ver, la consulta a la url nos devuelve en la propiedad "results" un arreglo de objetos, donde cada objeto contiene solo dos propiedades: **name** y **url**.

El valor de los atributos **url** representan el sub recurso de cada pokémon con el que podemos obtener la información detallada y específica de cada uno de ellos.

Para ver un ejemplo, ingresa a la url de "bulbasaur" y deberás recibir la data que se observa en la siguiente imagen:

```
4   {
5   ▶   "abilities": [↔],
23  ▶   "base_experience": 64,
24  ▶   "forms": [↔],
30  ▶   "game_indices": [↔],
172 ▶   "height": 7,
173 ▶   "held_items": [↔],
176   "id": 1,
177   "is_default": true,
178   "location_area_encounters": "https://pokeapi.co/api/v2/pokemon/1/encounters",
179 ▶   "moves": [↔],
10243 ▶   "name": "bulbasaur",
10244   "order": 1,
10245 ▶   "species": {↔},
10249 ▶   "sprites": {↔},
10408 ▶   "stats": [↔],
10458 ▶   "types": [↔],
10474   "weight": 69
10475 }
```

Imagen 15. Data específica de 1 pokémon.

Fuente: [PokeApi](https://pokeapi.co/)

El modelo HATEOAS ayuda a mejorar la navegación de los recursos que ofrecemos en nuestras API REST.

Entre sus características más reconocidas que acompañan este modelo de datos está la existencia de atributos que:

- Contienen la cantidad total de los recursos disponibles (**count**)
- Permiten la paginación de recursos (**next**, **previous**)
- Ofrecen un conjunto recortado de los recursos disponibles (**results**)

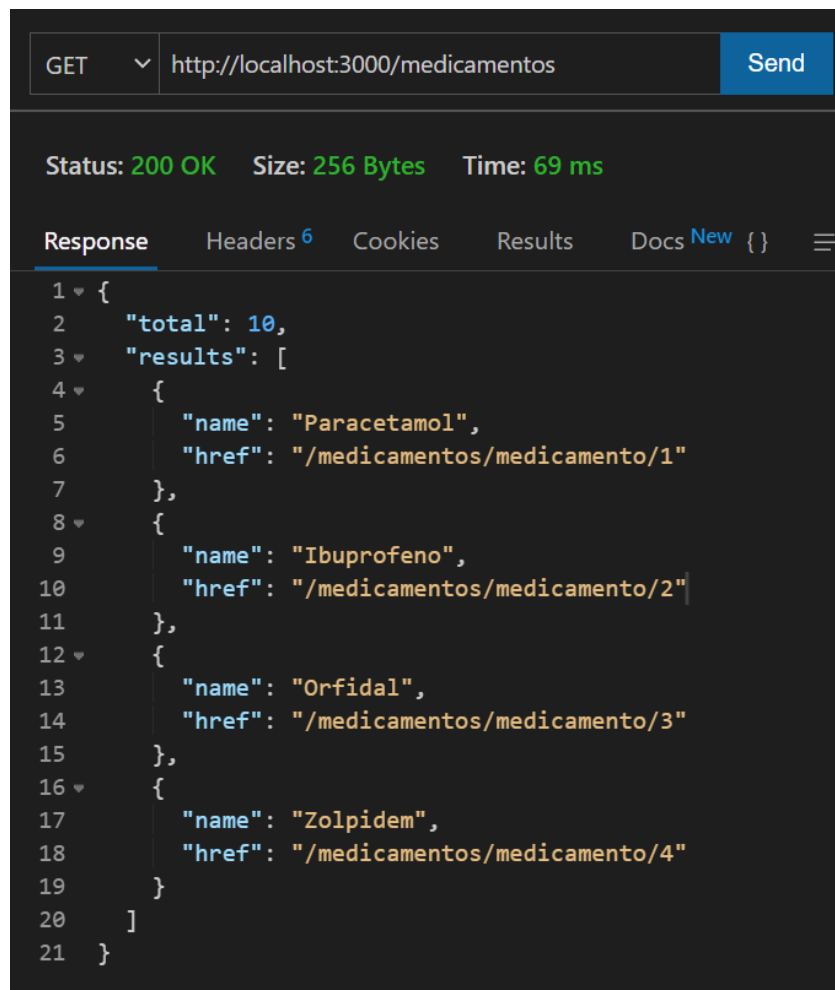
Implementemos HATEOAS en nuestro servidor de medicamentos creando una función llamada **prepararHATEOAS**.

```
const prepararHATEOAS = (medicamentos) => {  
  
  const results = medicamentos.map((m) => {  
    return {  
      name: m.nombre,  
      href: `/medicamentos/medicamento/${m.id}`,  
    }  
  }).slice(0, 4)  
  const total = medicamentos.length  
  const HATEOAS = {  
    total,  
    results  
  }  
  return HATEOAS  
}
```

Ahora utilicemos esta función dentro de la ruta **GET /medicamentos**:

```
app.get("/medicamentos", async (req, res) => {  
  const queryStrings = req.query;  
  const medicamentos = await obtenerMedicamentos(queryStrings);  
  const HATEOAS = await prepararHATEOAS(medicamentos)  
  res.json(HATEOAS);  
});
```

Consultemos nuevamente la ruta **GET /medicamentos** desde Thunder Client y veamos como se presenta ahora la información.



```
GET http://localhost:3000/medicamentos Send

Status: 200 OK Size: 256 Bytes Time: 69 ms

Response Headers 6 Cookies Results Docs New {}

1 {
2   "total": 10,
3   "results": [
4     {
5       "name": "Paracetamol",
6       "href": "/medicamentos/medicamento/1"
7     },
8     {
9       "name": "Ibuprofeno",
10      "href": "/medicamentos/medicamento/2"
11    },
12    {
13      "name": "Orfidal",
14      "href": "/medicamentos/medicamento/3"
15    },
16    {
17      "name": "Zolpidem",
18      "href": "/medicamentos/medicamento/4"
19    }
20  ]
21 }
```

Imagen 16. HATEOAS aplicado a **GET /medicamentos**
Fuente: Desafío Latam



¡Manos a la obra! - Implementa HATEOAS

Implementa el modelo de datos HATEOAS para obtener el personal de la farmacia

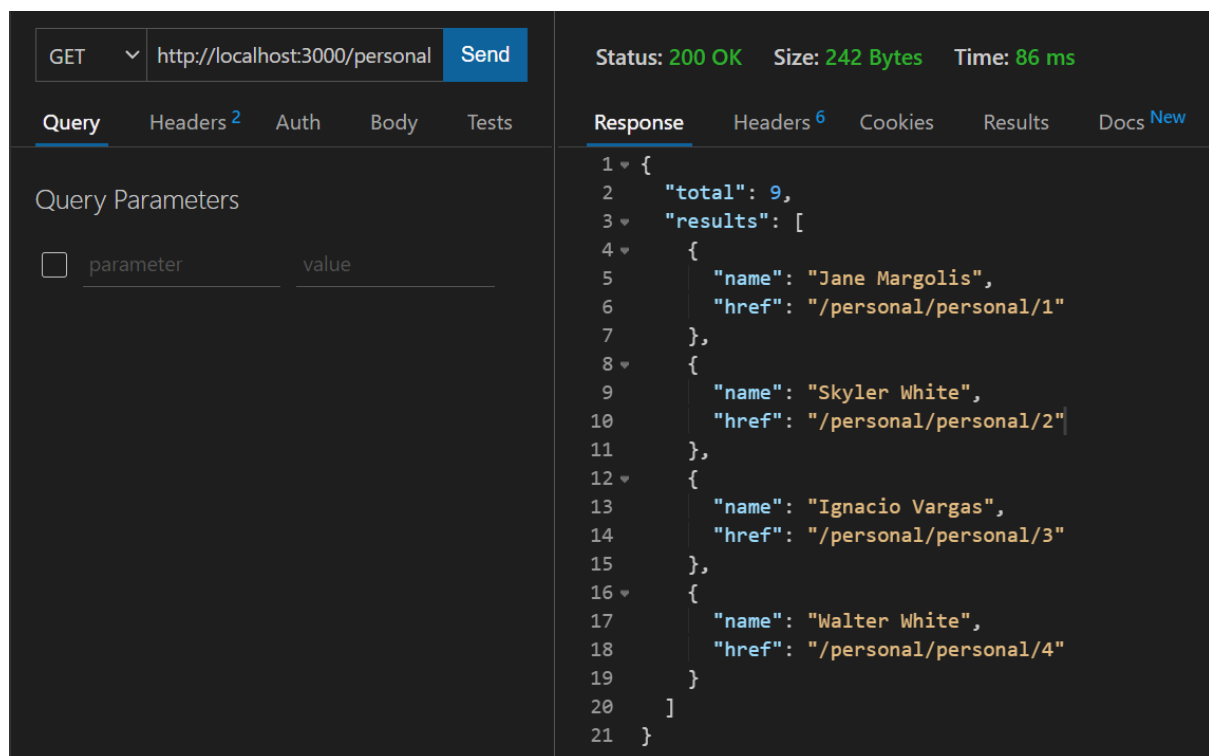


Imagen 16. HATEOAS aplicado a **GET /personal**

Fuente: Desafío Latam

La ruta por defecto

Es posible que una aplicación cliente consulte a nuestro servidor escribiendo una URL errada o que simplemente no coincida con ninguna de las rutas creadas.

Esto provocará una respuesta automática de **Express** indicando que la ruta y el método especificado en la consulta no fue encontrada

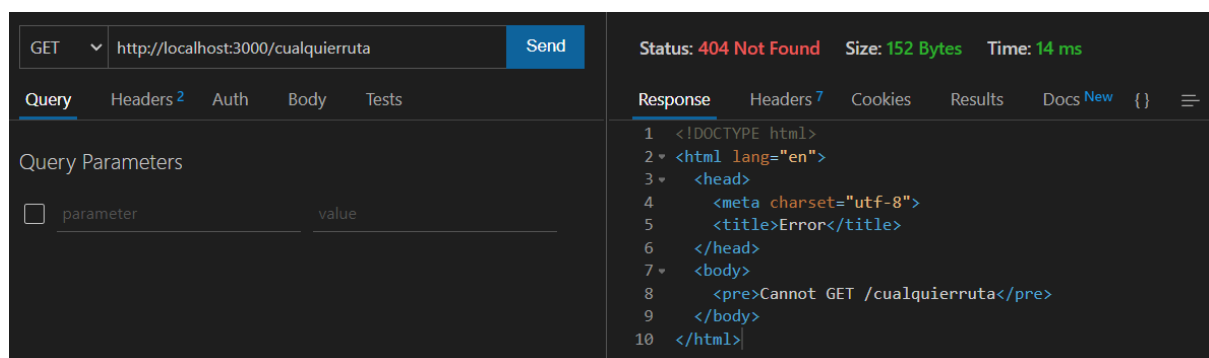


Imagen 14. Respuesta por defecto de Express

Fuente: Desafío Latam

Podemos personalizar esta respuesta agregando la ruta por defecto:

```
app.get("*", (req, res) => {  
  res.status(404).send("Esta ruta no existe")  
})
```

Con el símbolo asterisco(*) podremos reconocer cualquier URL recibida cuando ninguna de nuestras rutas coincidan.

Esta ruta debe estar declarada **después** de todas las anteriores, al final del código de nuestro servidor.

Ahora, al consultar nuevamente una ruta inexistente, obtendremos lo siguiente:

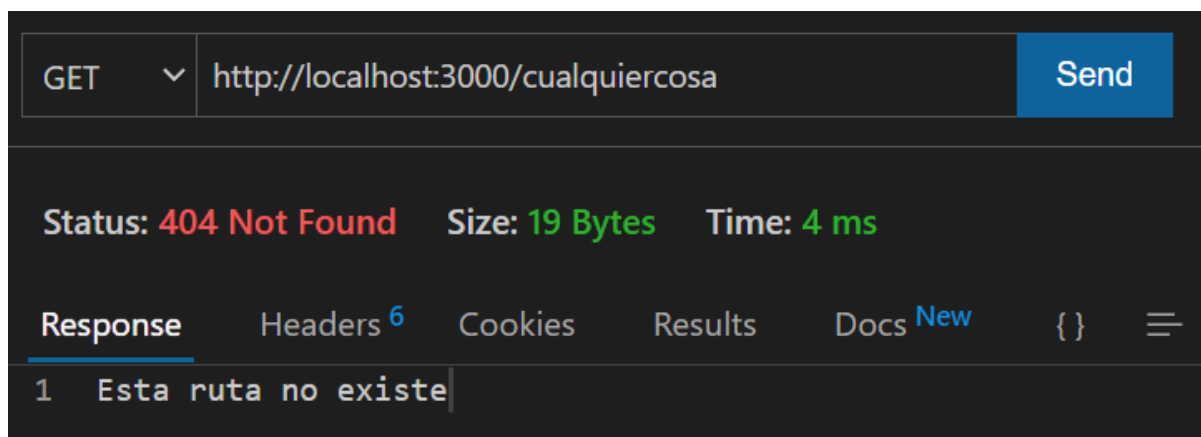


Imagen 15. Respuesta personalizada en una ruta inexistente
Fuente: Desafío Latam



¡Manos a la obra! - Servidor personal

En el servidor de personal:

- Agrega la ruta por defecto
- Prueba la ruta por defecto con Thunder Client

Preguntas

1. ¿Qué cambio tenemos que hacer en este código para que la primera página sea la página 1 y no la página 0?

```
const obtenerMedicamentos = async ({ limits = 10, order_by = "id_ASC", page  
= 0 }) => {
```

```
const [campo, direccion] = order_by.split("_")
const offset = page * limits

const formattedQuery = format('SELECT * FROM medicamentos order by %s %s
LIMIT %s OFFSET %s', campo, direccion, limits, offset);

pool.query(formattedQuery);

const { rows: medicamentos } = await pool.query(formattedQuery)
return medicamentos
}
```

2. ¿Qué deberíamos modificar en el código anterior para tener páginas de 100 registros?
3. ¿Cuál es el problema con este código?

```
// Agregamos la función obtenerMedicamentos
const obtenerMedicamentos = async ({ limits }) => {
  let consulta = "SELECT * FROM medicamentos"
  if (limits) consulta += " LIMIT " + limits
  const { rows: medicamentos } = await pool.query(consulta)
  return medicamentos
}
```