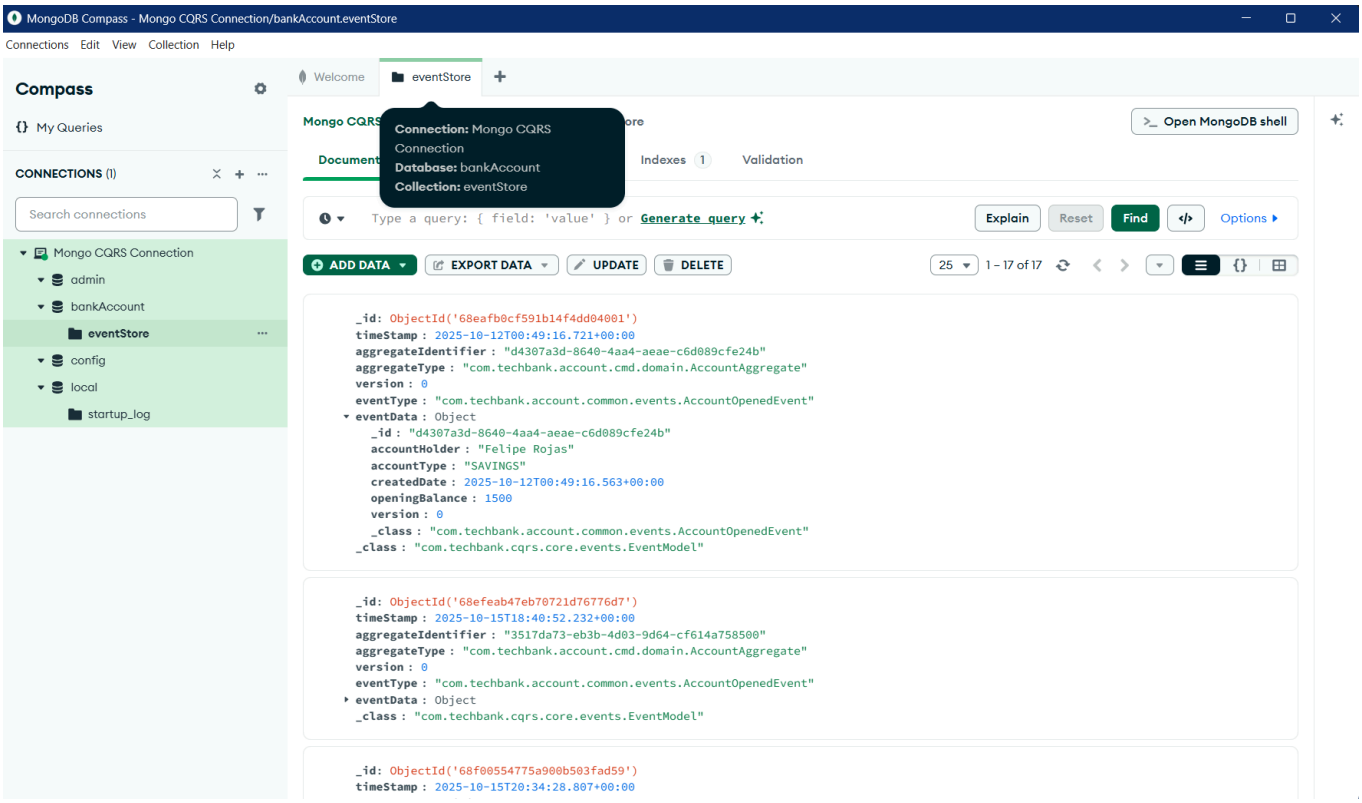
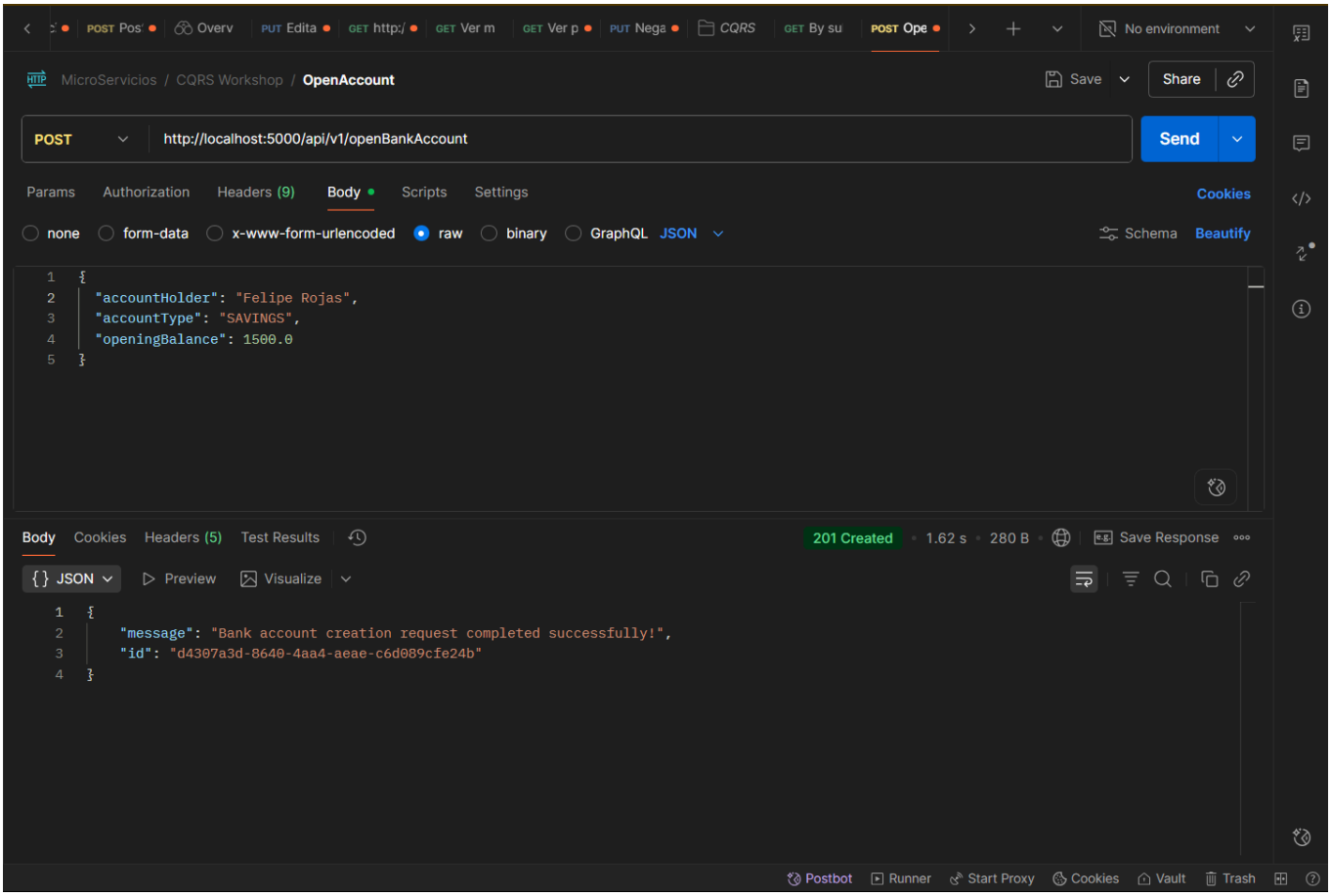
Felipe Rojas Prado - A00393918

Taller de Micro Servicios

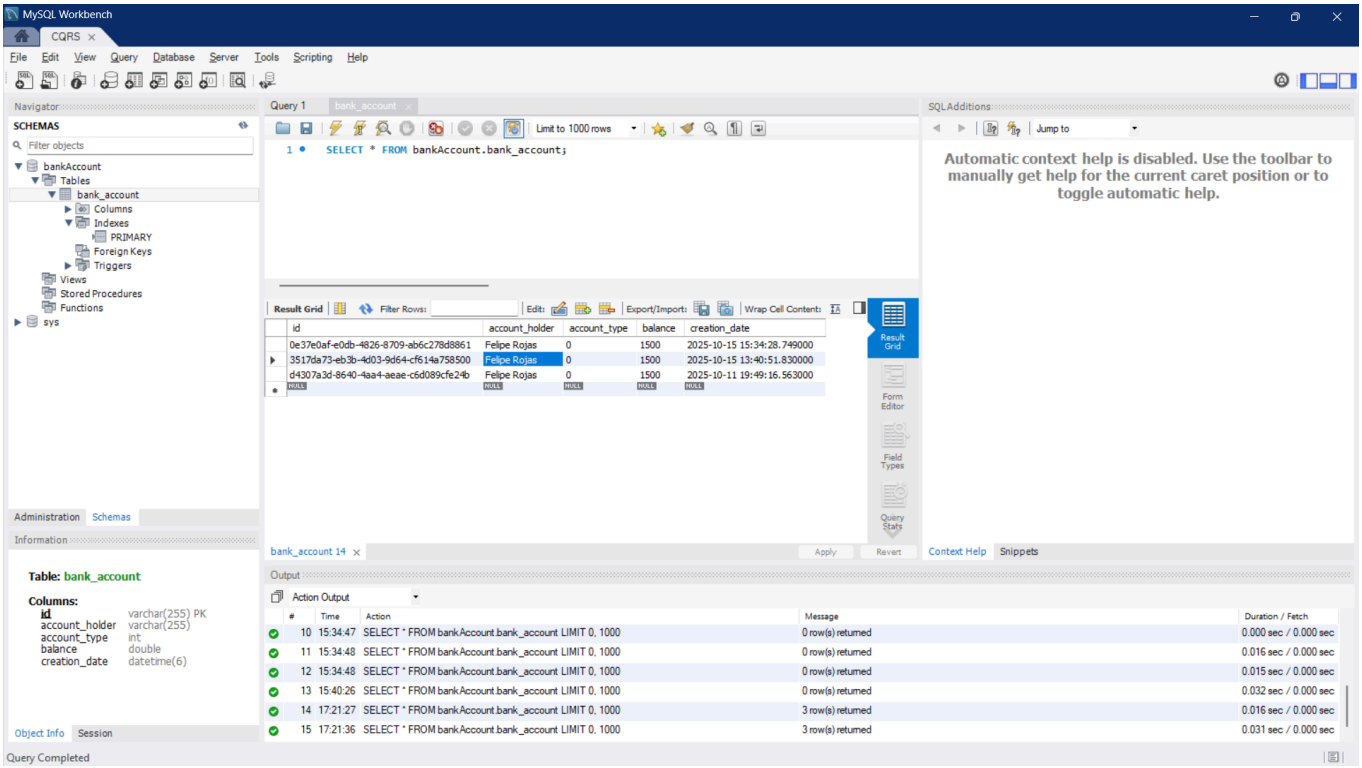
Capturas de pantalla

Mongo Compass

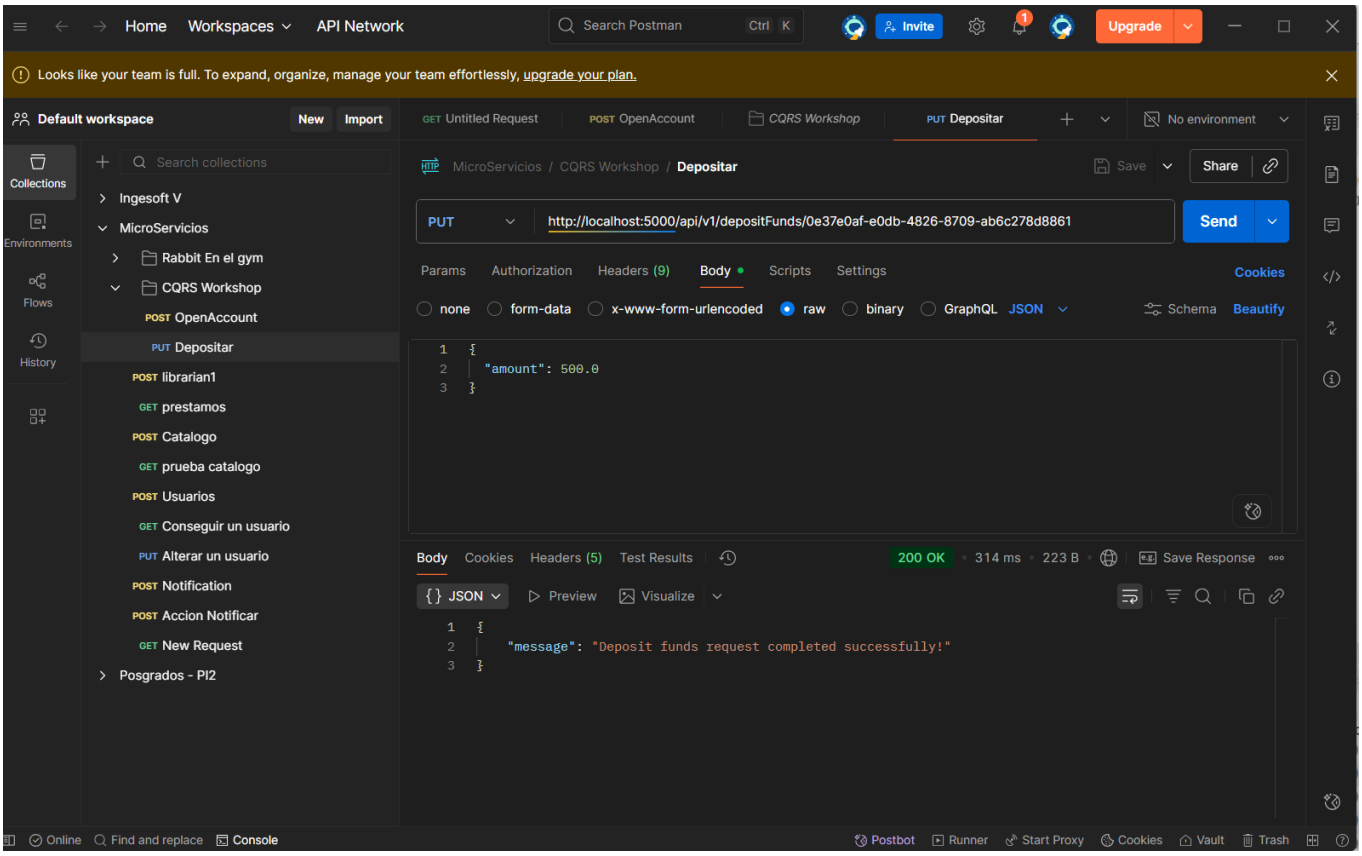




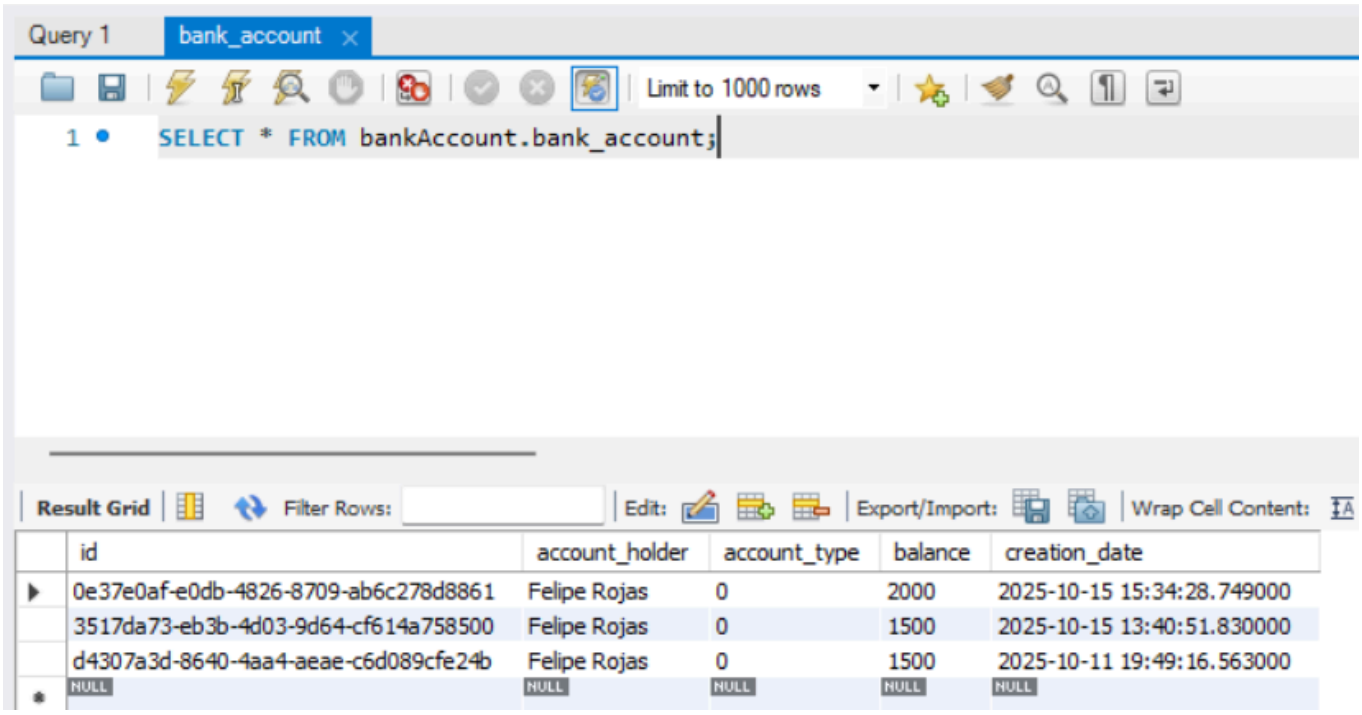
Revision en MySQL Workbench:



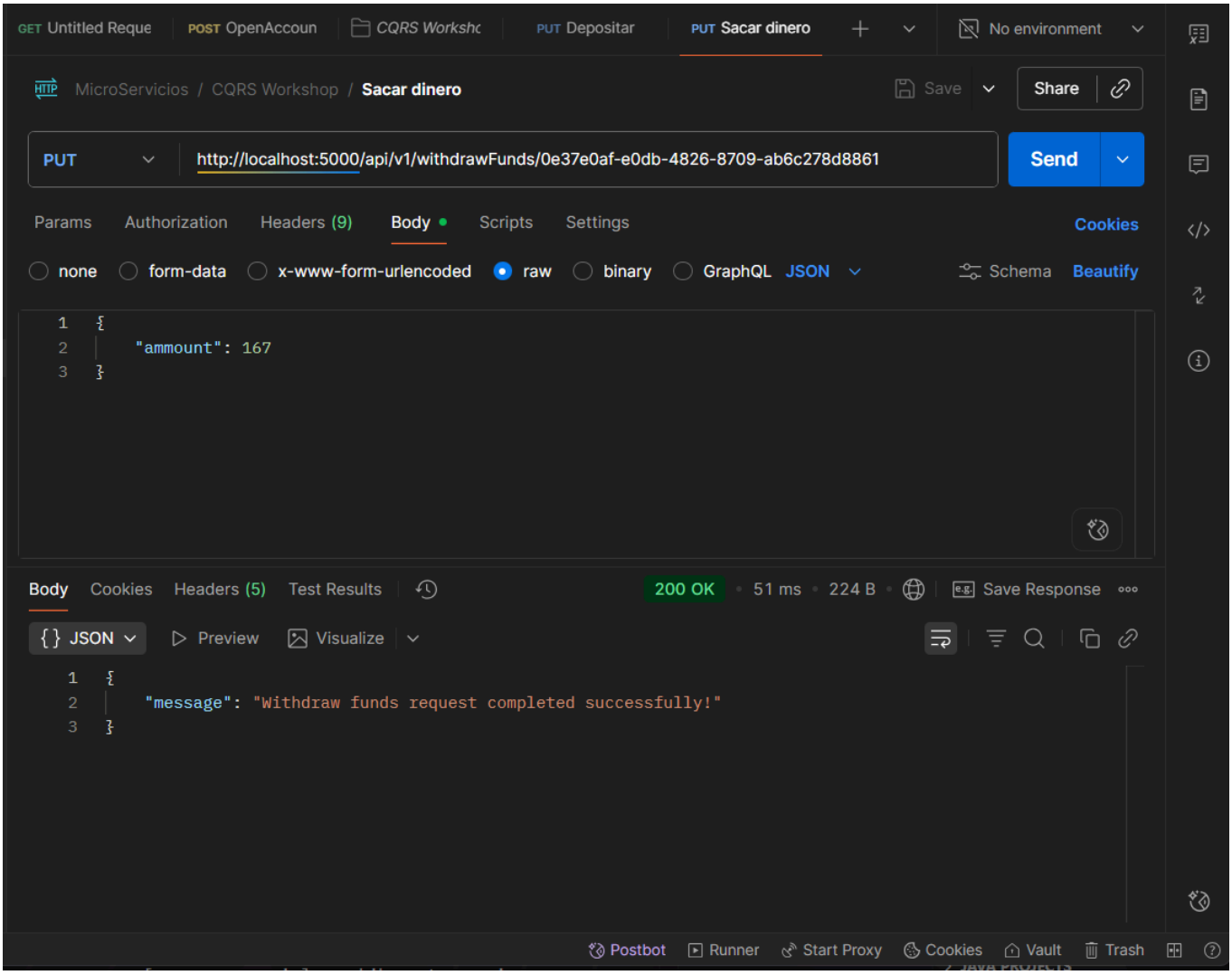
Deposito



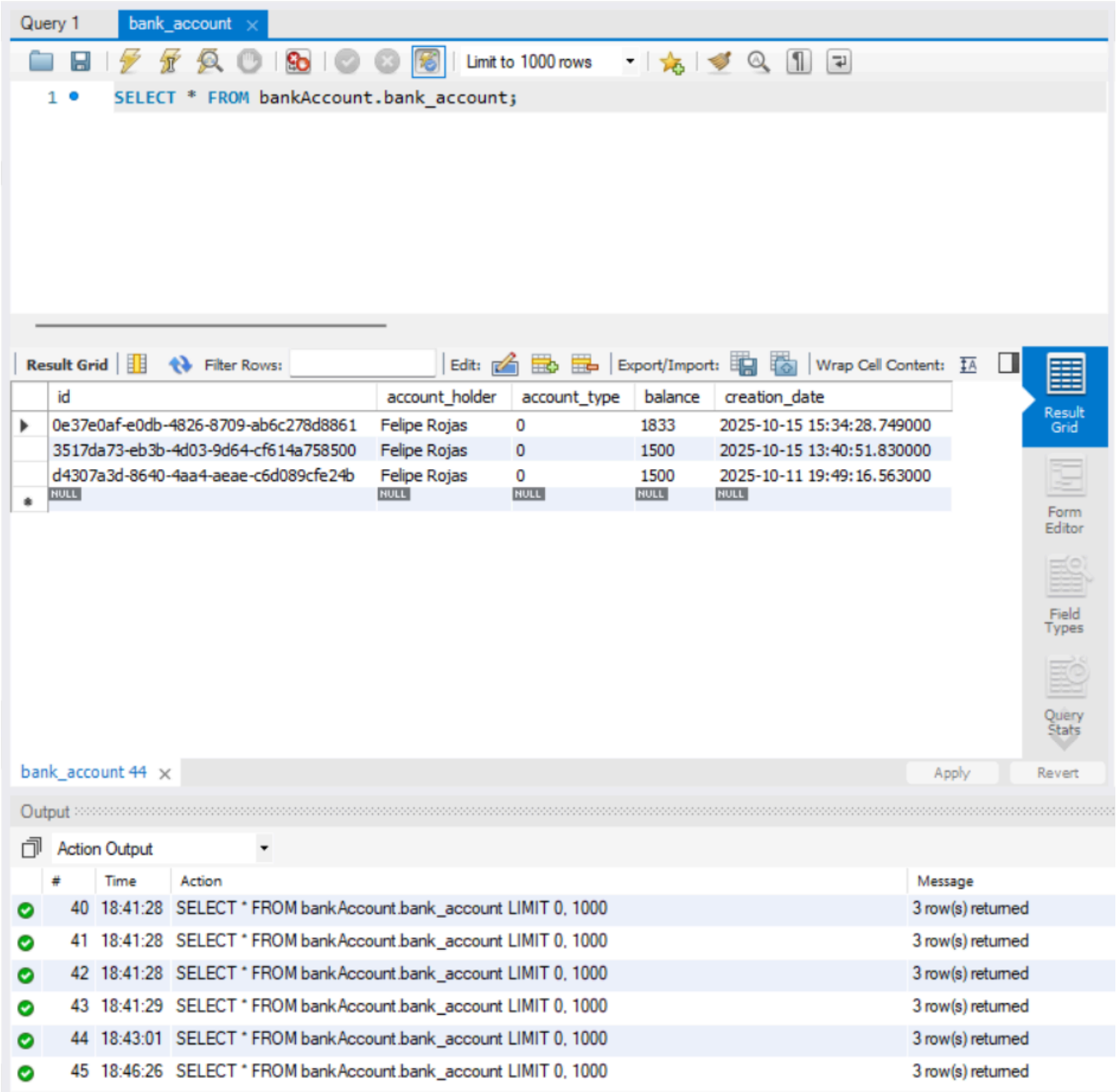
Cambio del balance:



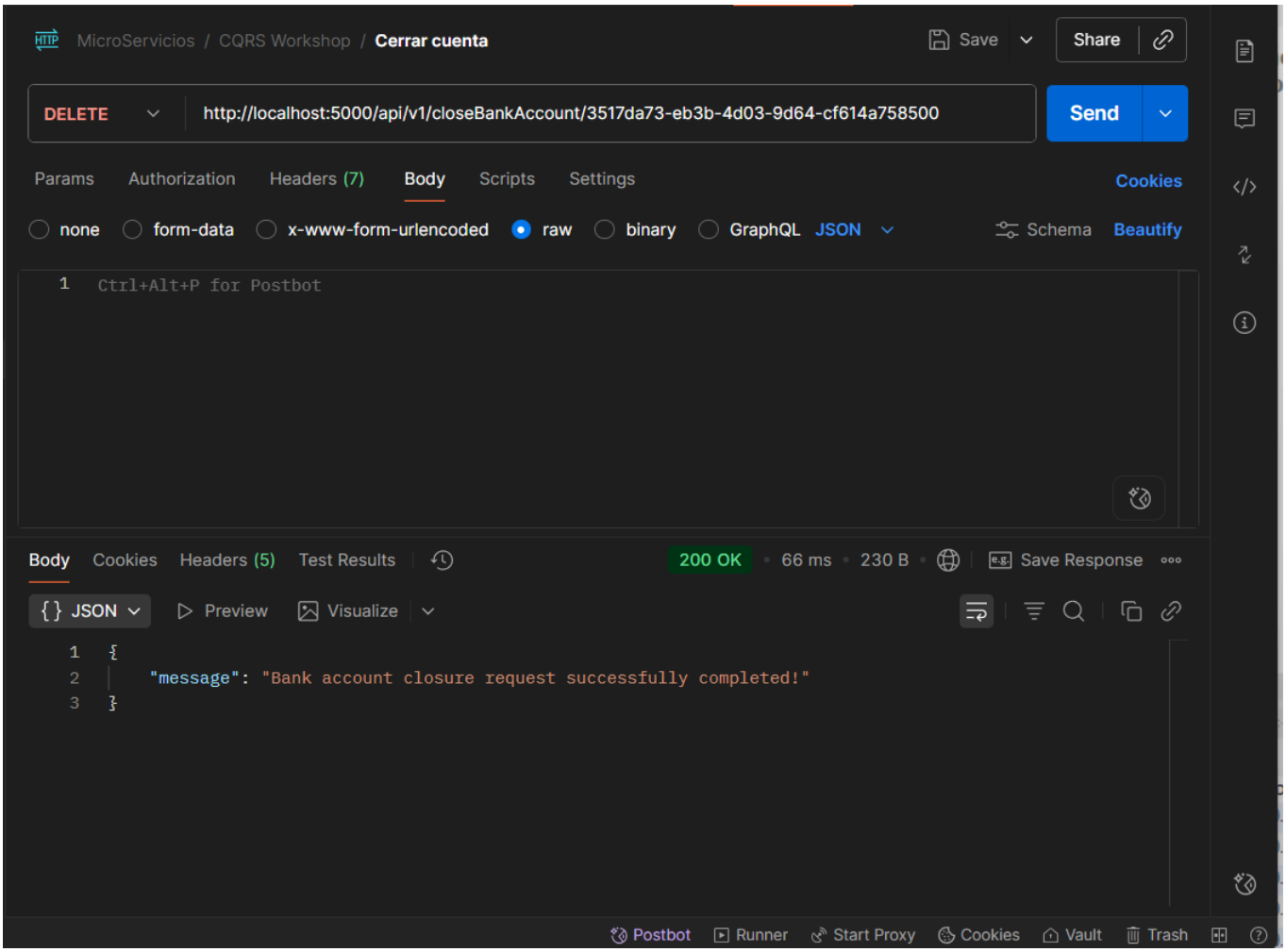
Sacar dinero



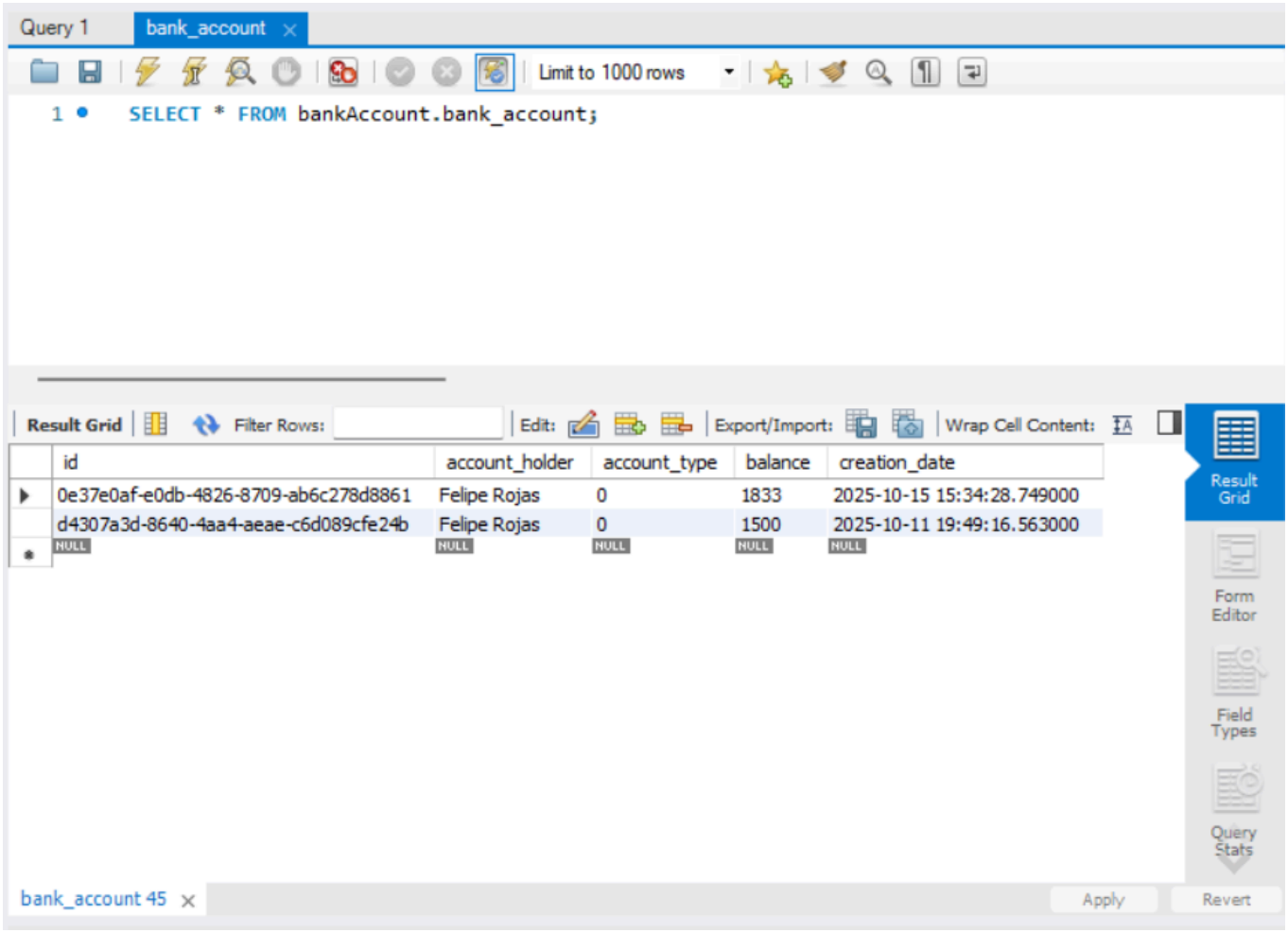
Retiro del dinero:



Cerrar cuenta

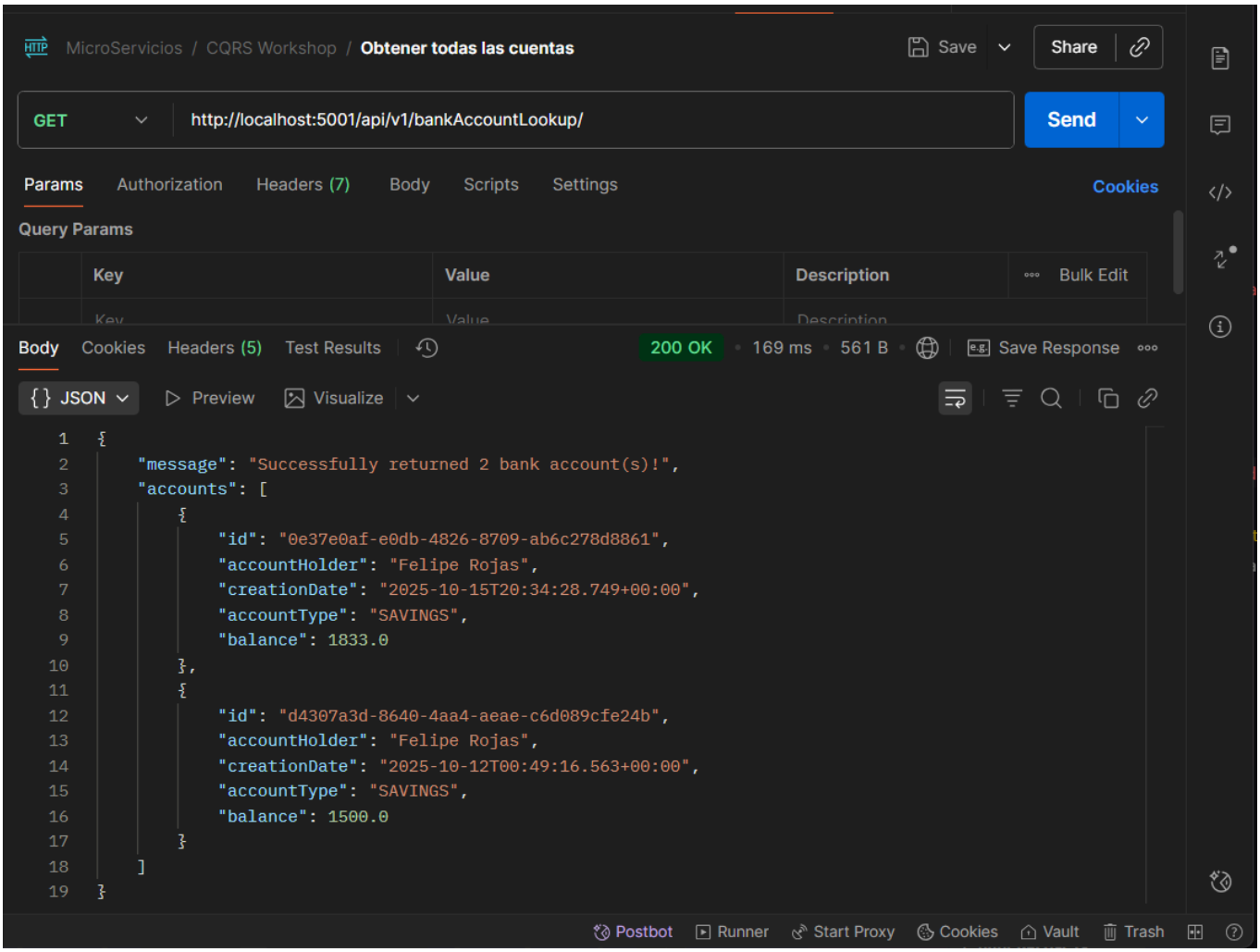

MicroServicios / CQRS Workshop / Cerrar cuenta
Save
Share
DELETE
http://localhost:5000/api/v1/closeBankAccount/3517da73-eb3b-4d03-9d64-cf614a758500
Send
Params
Authorization
Headers (7)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Schema
Beautify
1 Ctrl+Alt+P for Postbot
Body
Cookies
Headers (5)
Test Results
200 OK
66 ms
230 B
Save Response
{} JSON
Preview
Visualize
1 {
2 "message": "Bank account closure request successfully completed!"
3 }
Postbot
Runner
Start Proxy
Cookies
Vault
Trash

Cierre exitoso de cuenta

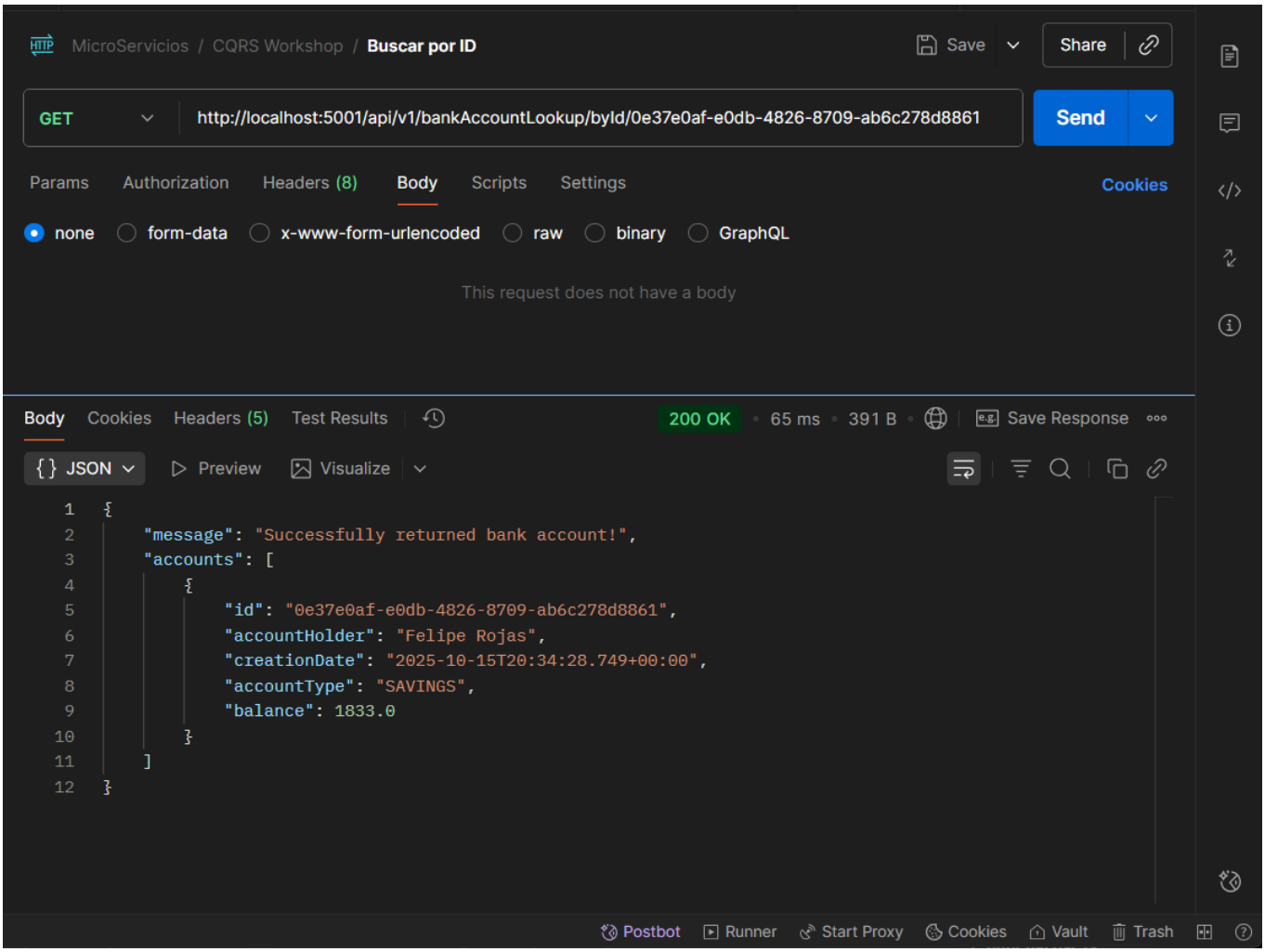


Peticiones GET

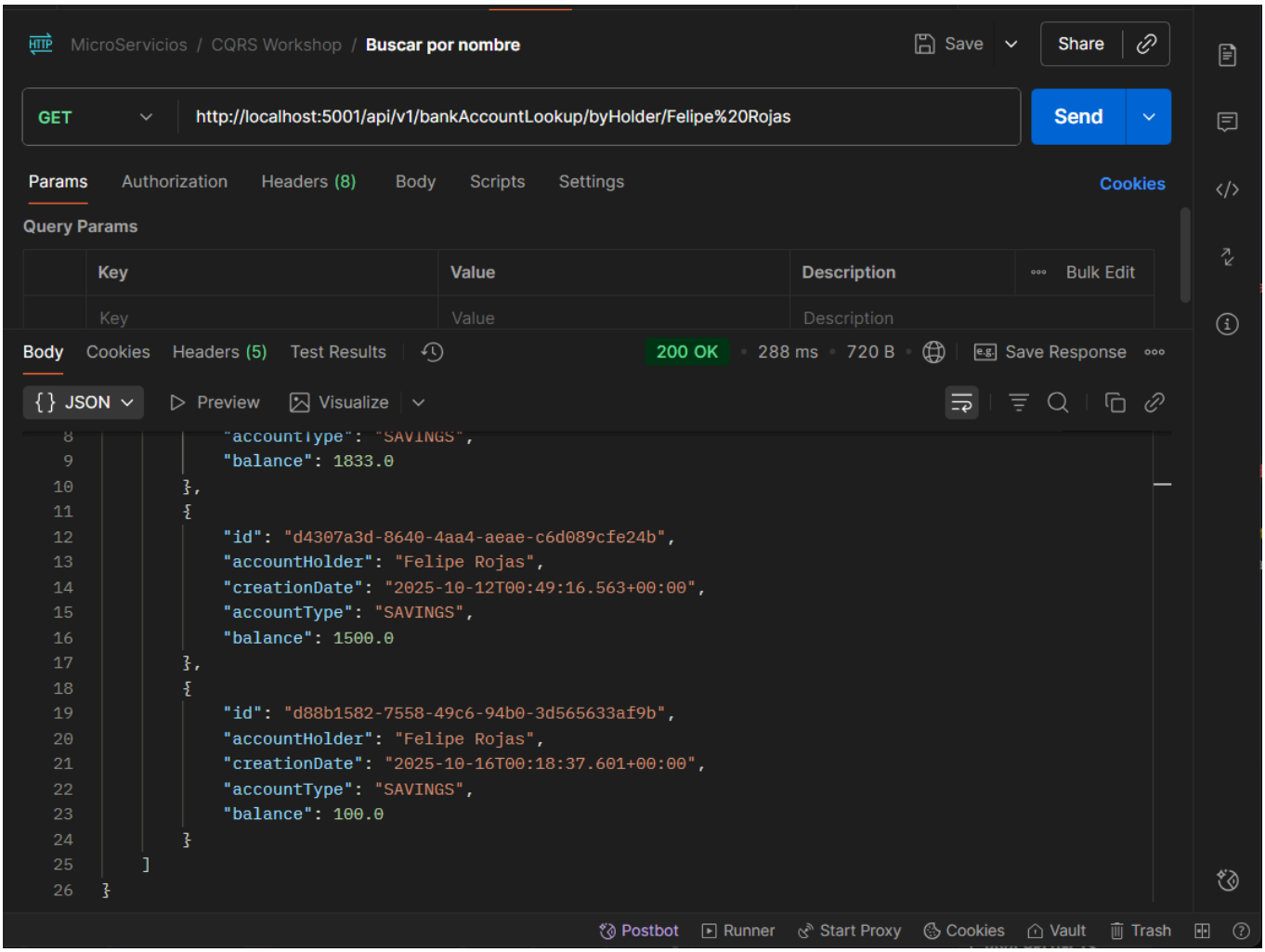
Obtener todas las cuentas



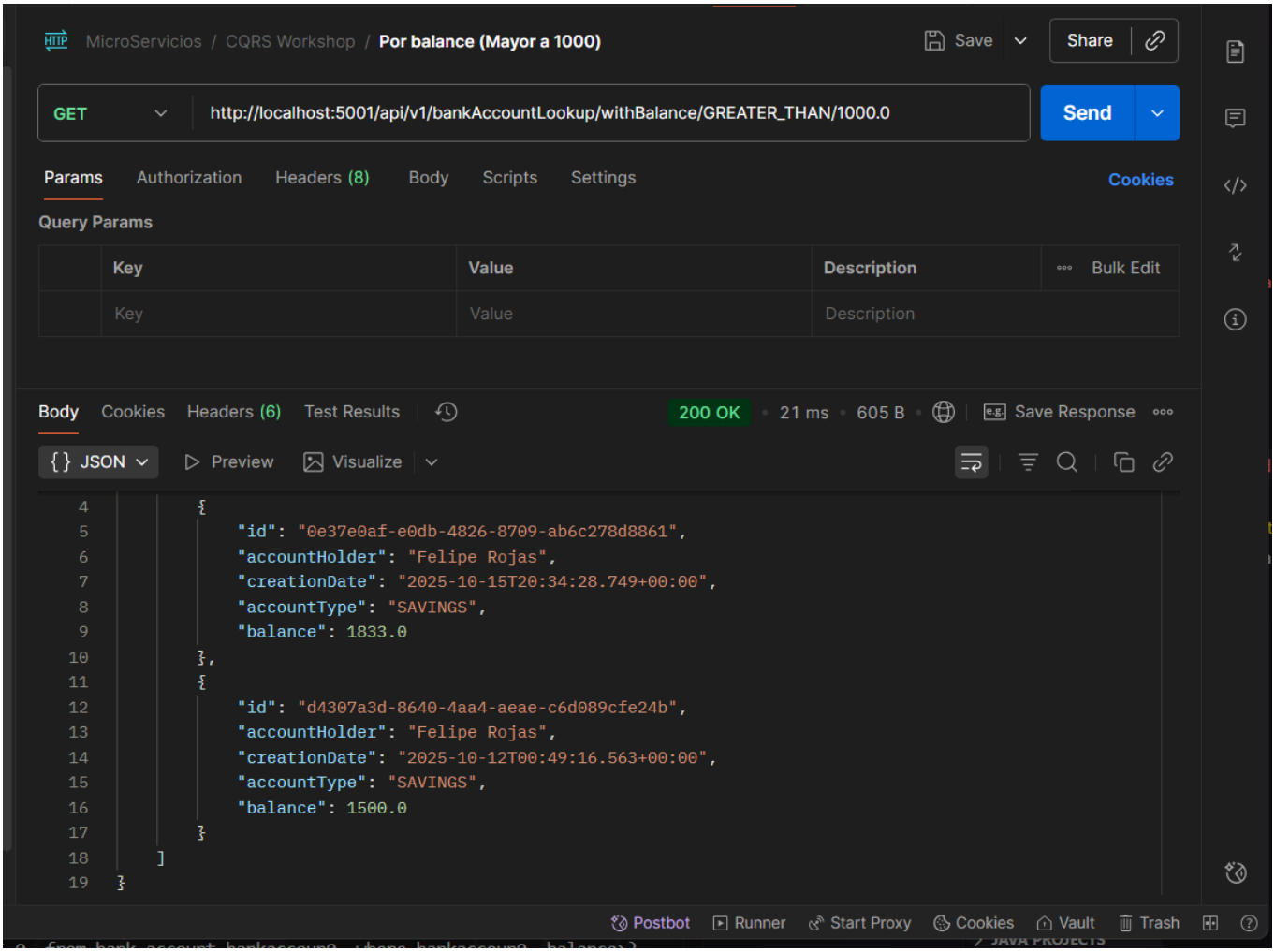
Buscar por ID



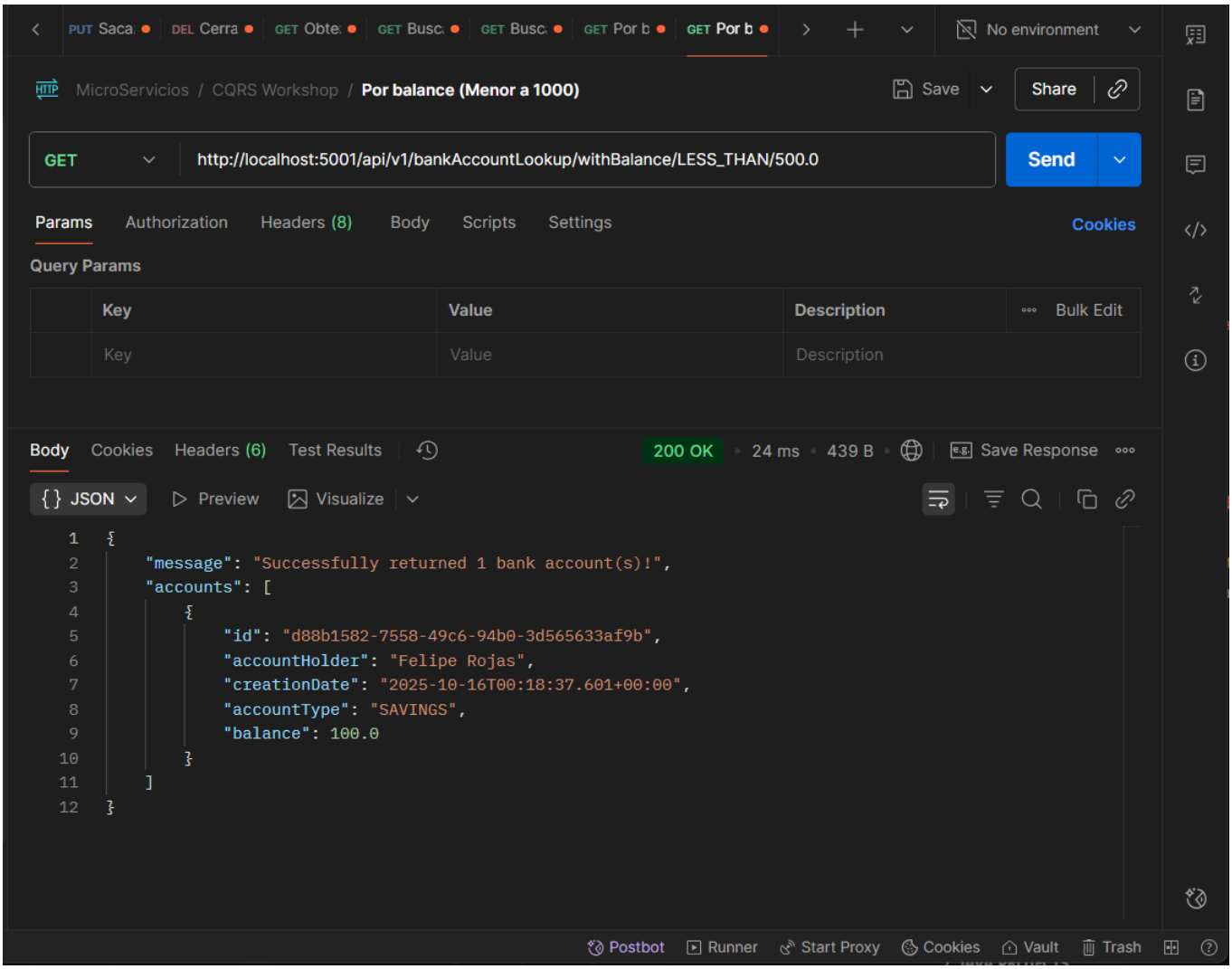
Buscar por nombre



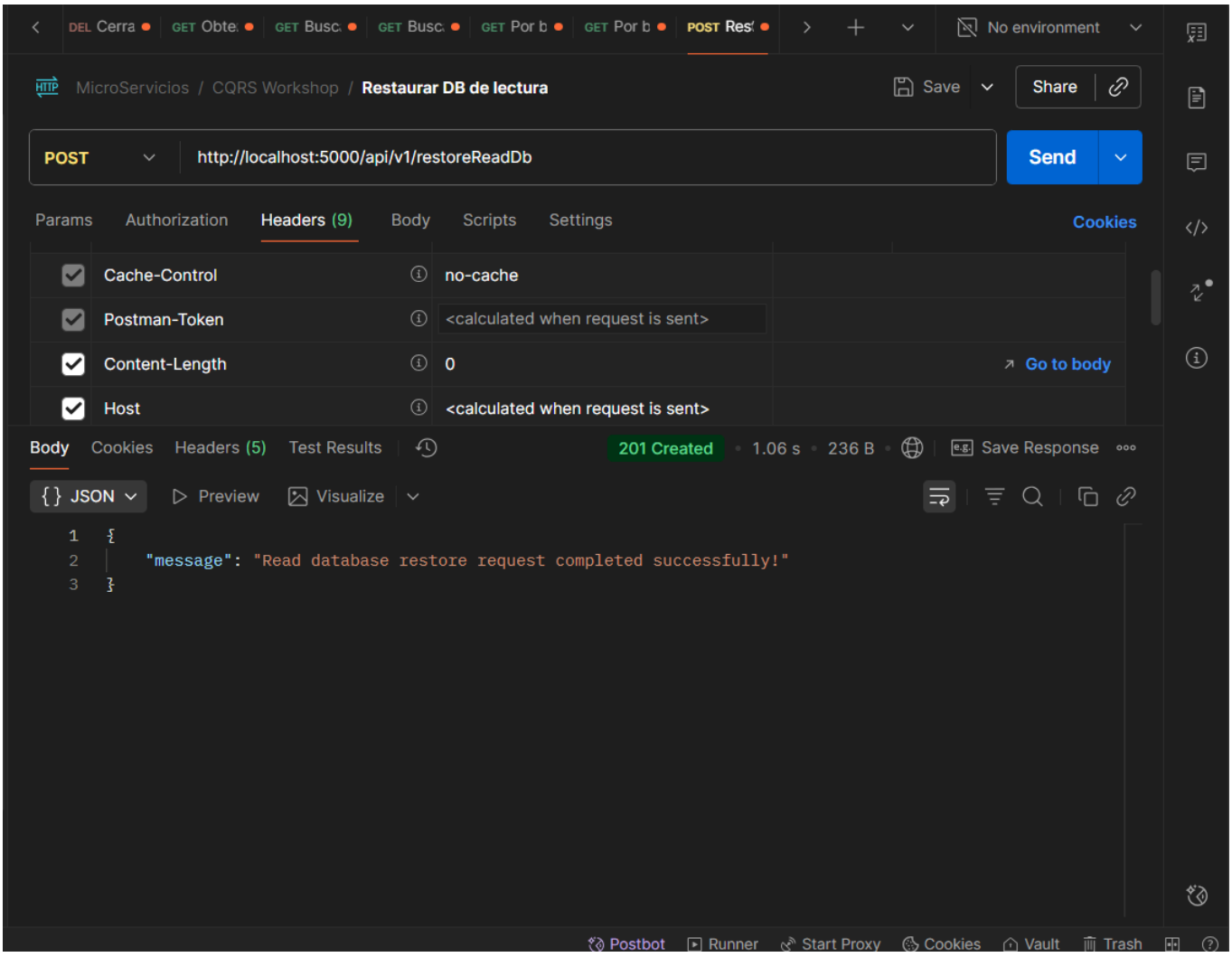
Buscar por balance mayor a 1000



Buscar por balance menor a 1000

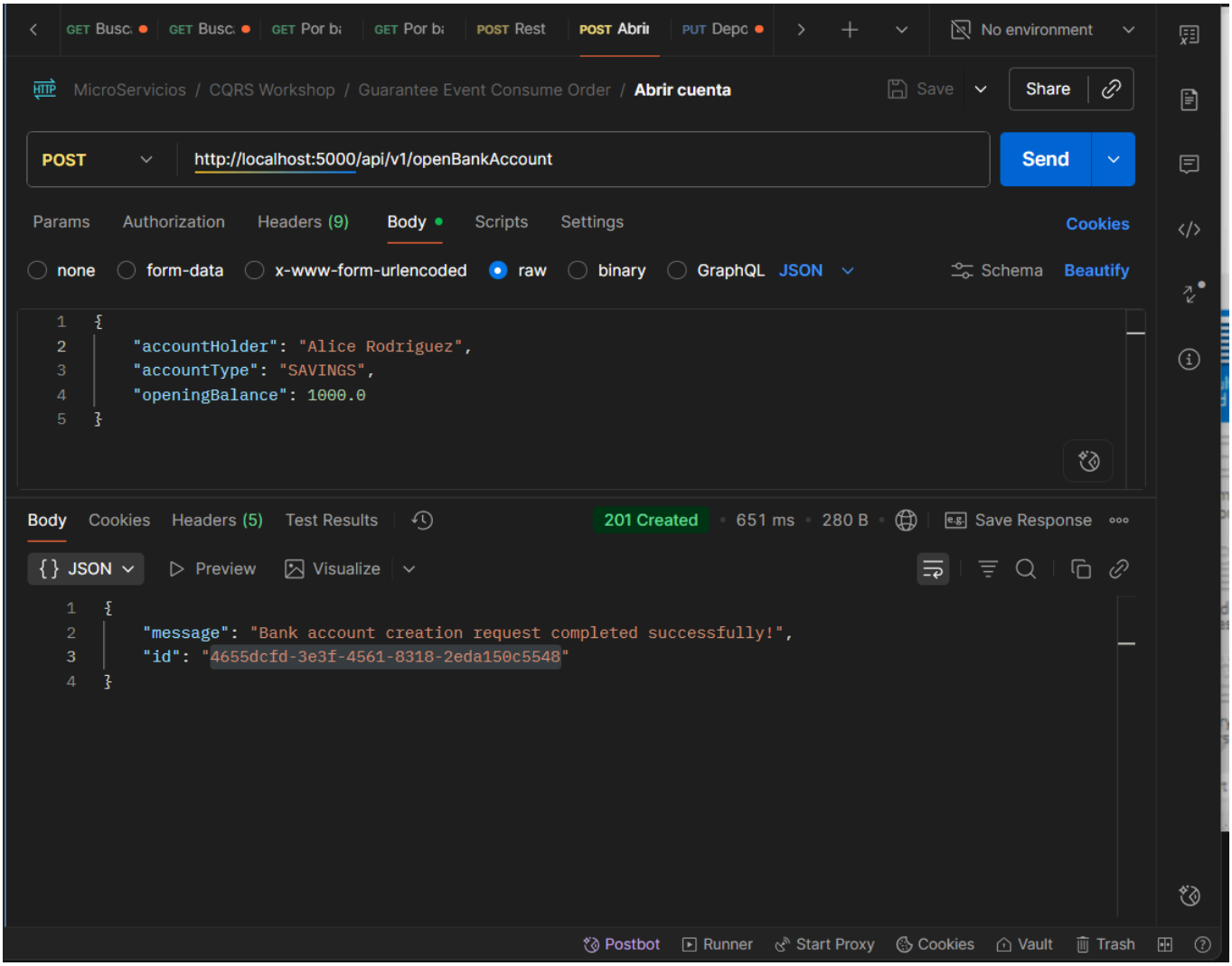


Restaurar la DB de lectura

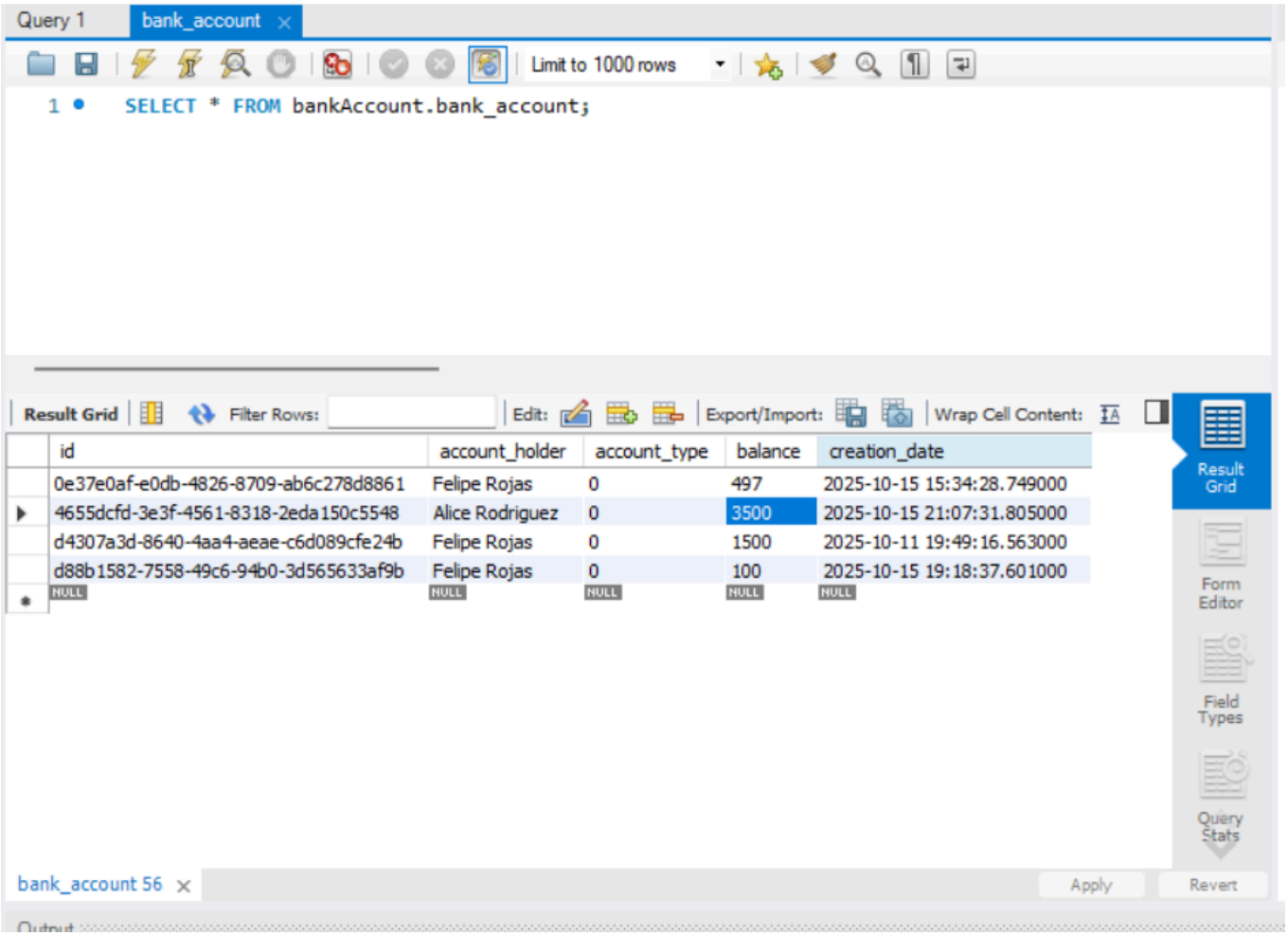


Parte 2

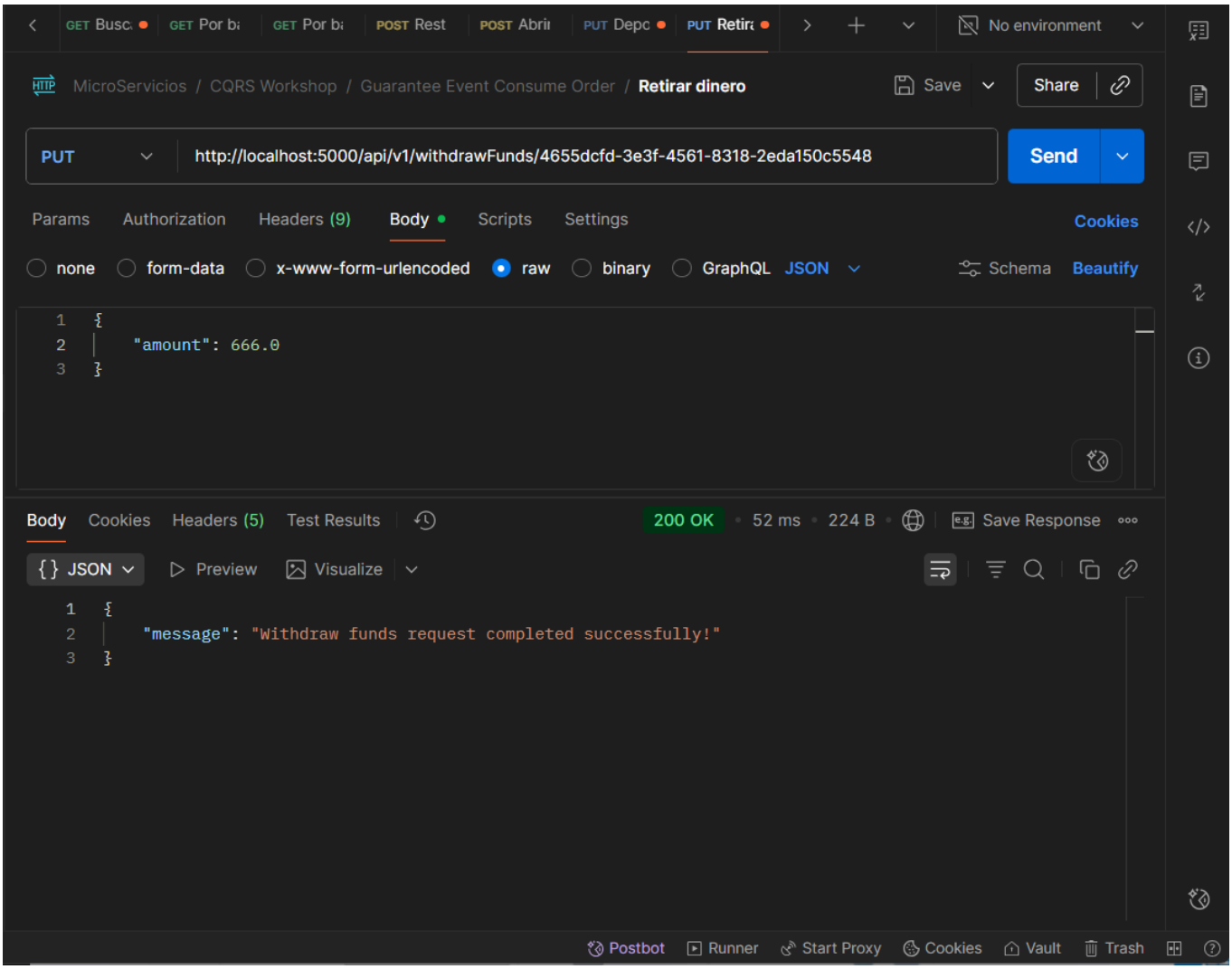
Crear cuenta

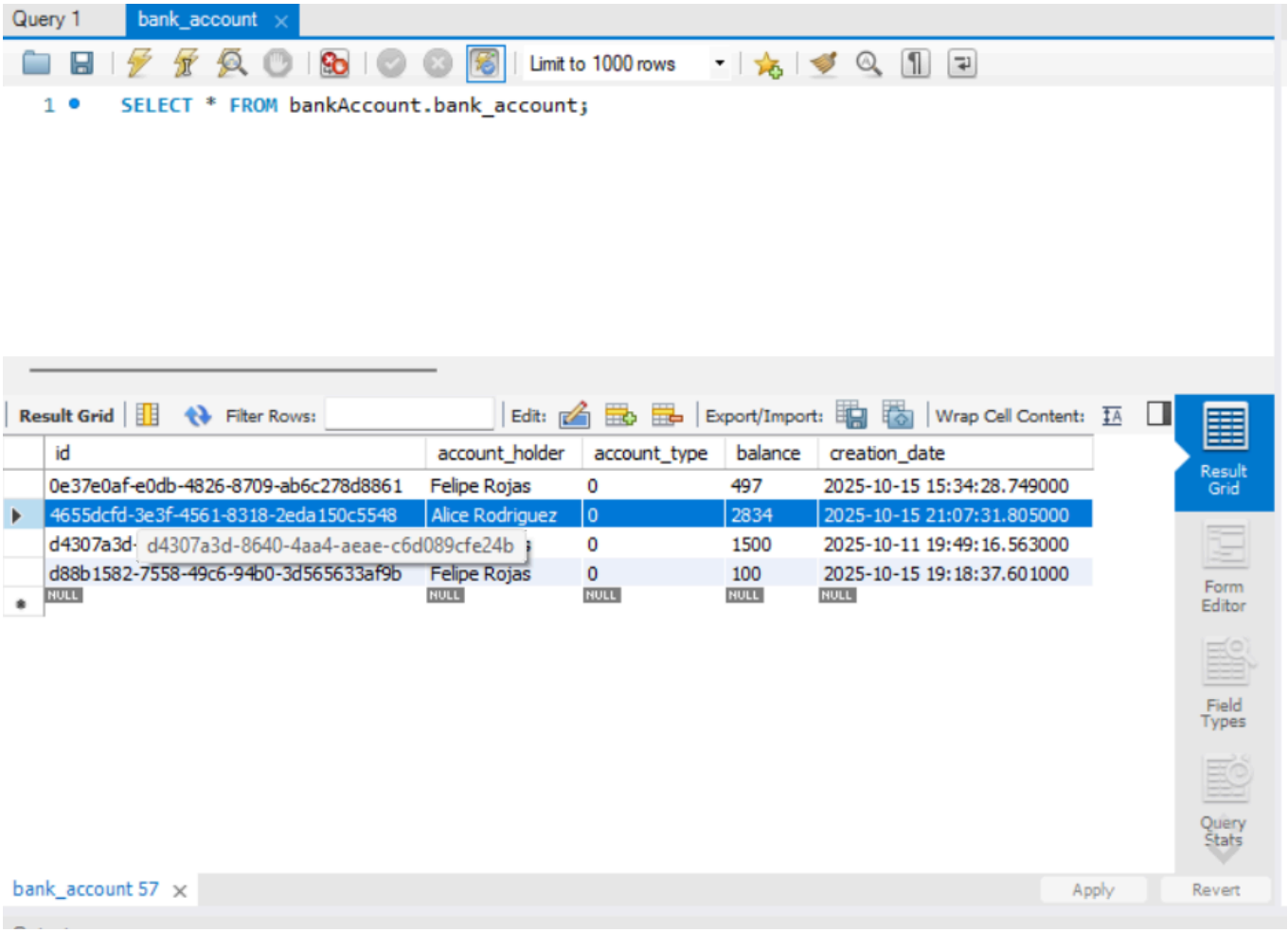


Depositar dinero

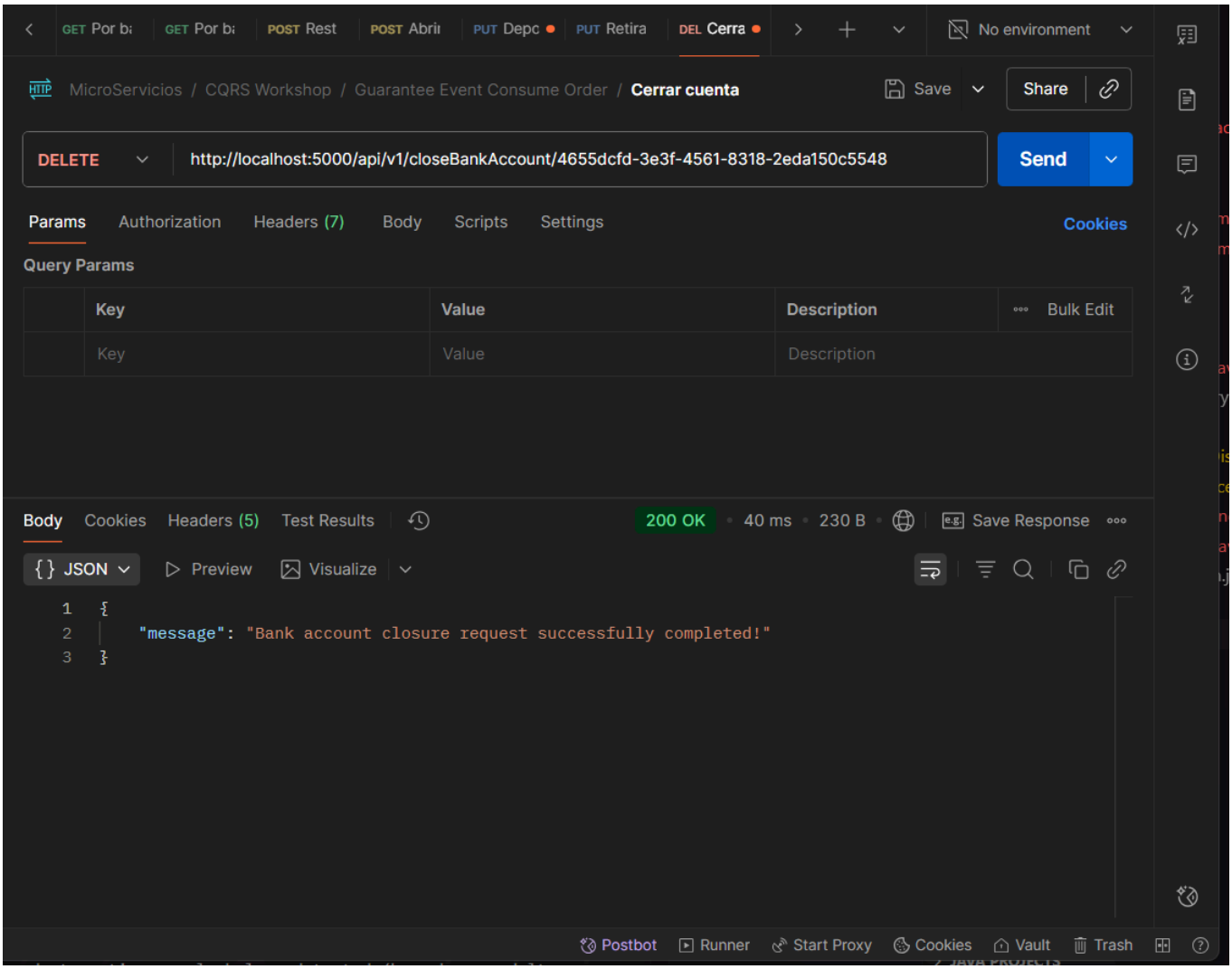


Retirar dinero

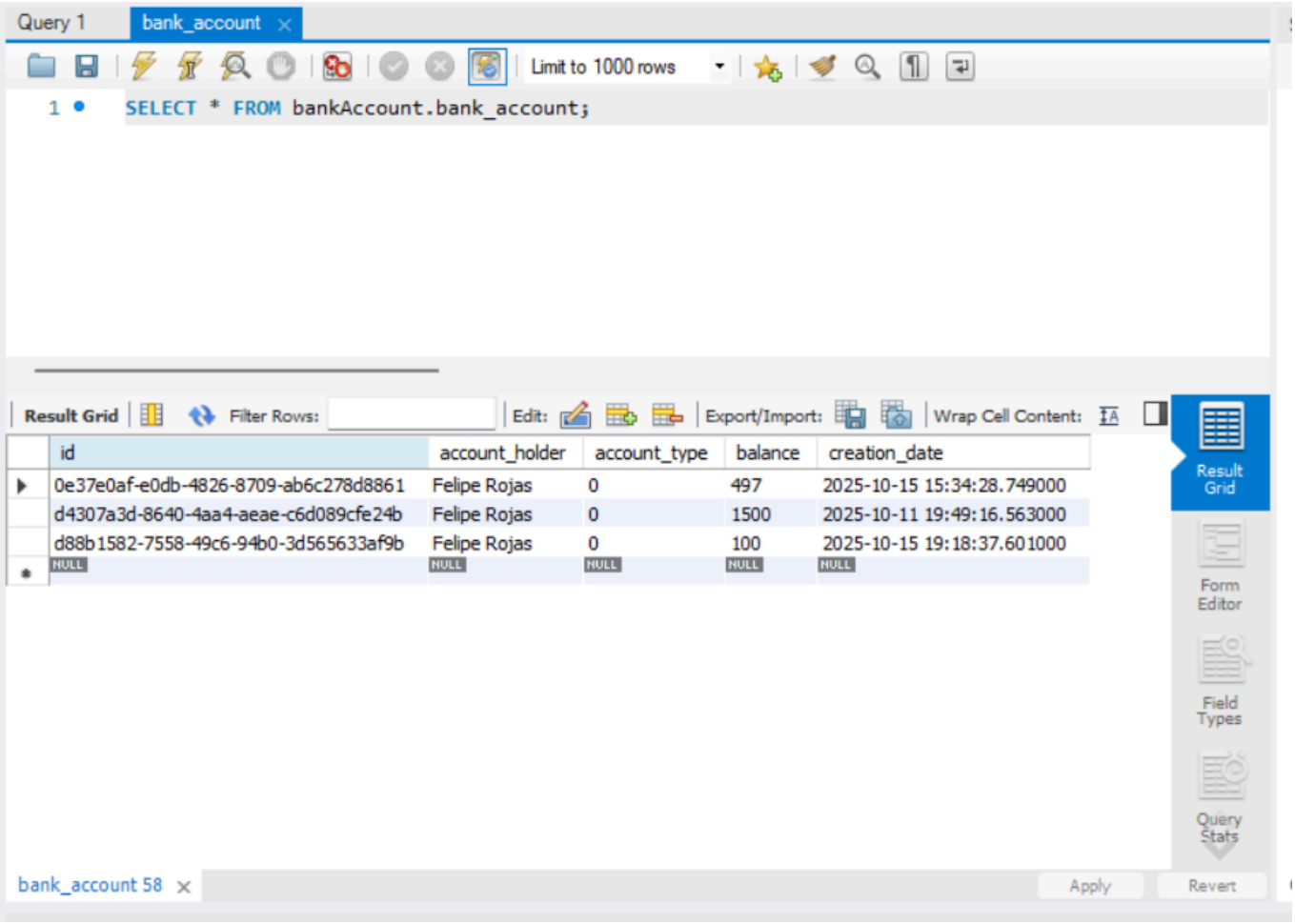

GET Busc.
GET Por b
GET Por b
POST Rest
POST Abri
PUT Depc
PUT Retir
No environment
MicroServicios / CQRS Workshop / Guarantee Event Consume Order / Retirar dinero
Save
Share
PUT
http://localhost:5000/api/v1/withdrawFunds/4655dcfd-3e3f-4561-8318-2eda150c5548
Send
Params
Authorization
Headers (9)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Schema
Beautify
{
"amount": 666.0
}
Body
Cookies
Headers (5)
Test Results
200 OK
52 ms
224 B
Save Response
{} JSON
Preview
Visualize
{
"message": "Withdraw funds request completed successfully!"
}
Postbot
Runner
Start Proxy
Cookies
Vault
Trash



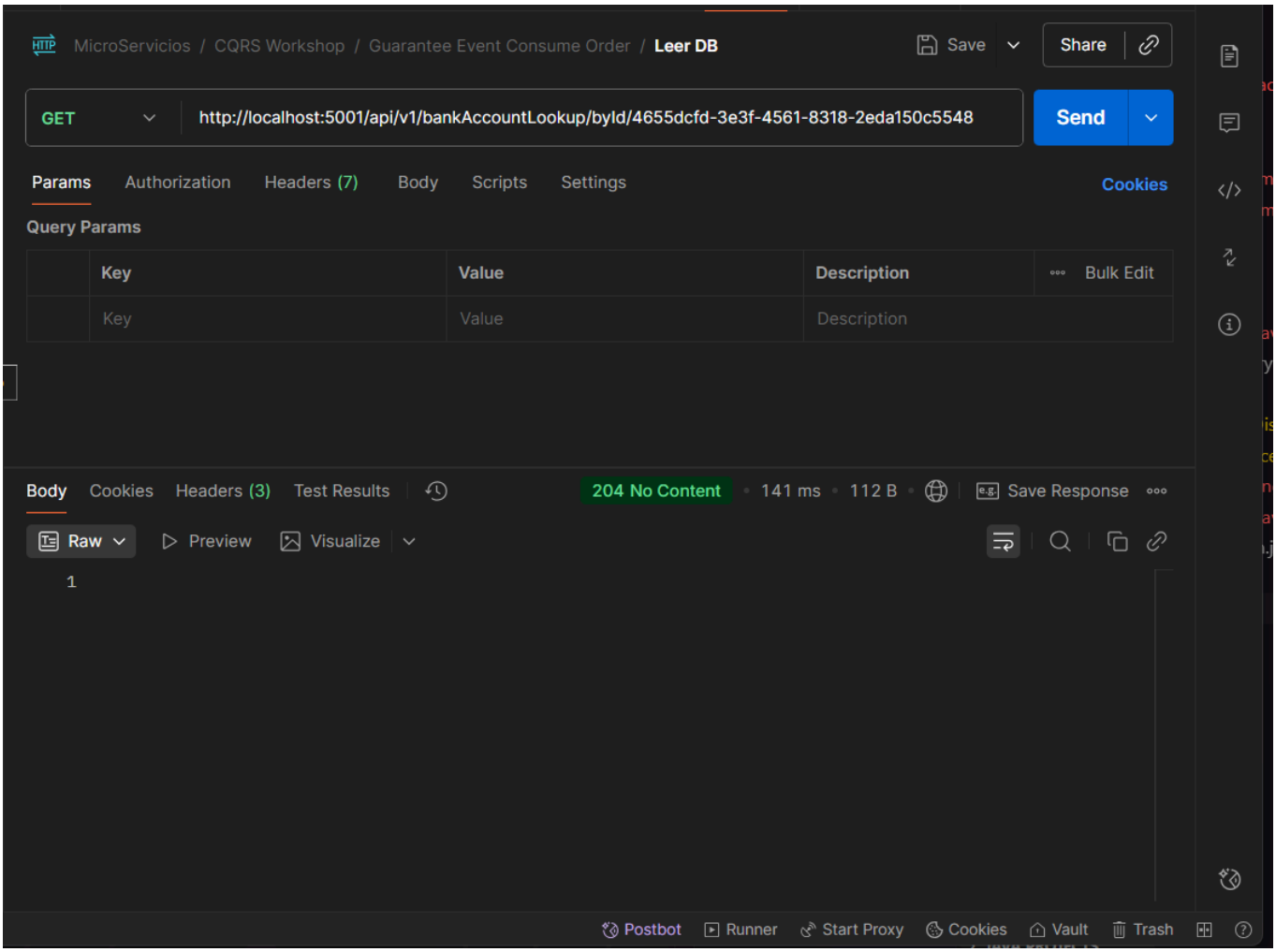
Borrar cuenta


GET Por bi
GET Por bi
POST Rest
POST Abrir
PUT Depc
PUT Retira
DEL Cerra
No environment
MicroServicios / CQRS Workshop / Guarantee Event Consume Order / Cerrar cuenta
Save
Share
DELETE
http://localhost:5000/api/v1/closeBankAccount/4655dcfd-3e3f-4561-8318-2eda150c5548
Send
Params
Authorization
Headers (7)
Body
Scripts
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Key
Value
Description
Body
Cookies
Headers (5)
Test Results
200 OK
40 ms
230 B
Save Response
JSON
Preview
Visualize
{
"message": "Bank account closure request successfully completed!"
}
Postbot
Runner
Start Proxy
Cookies
Vault
Trash

Cierre de cuenta satisfactorio

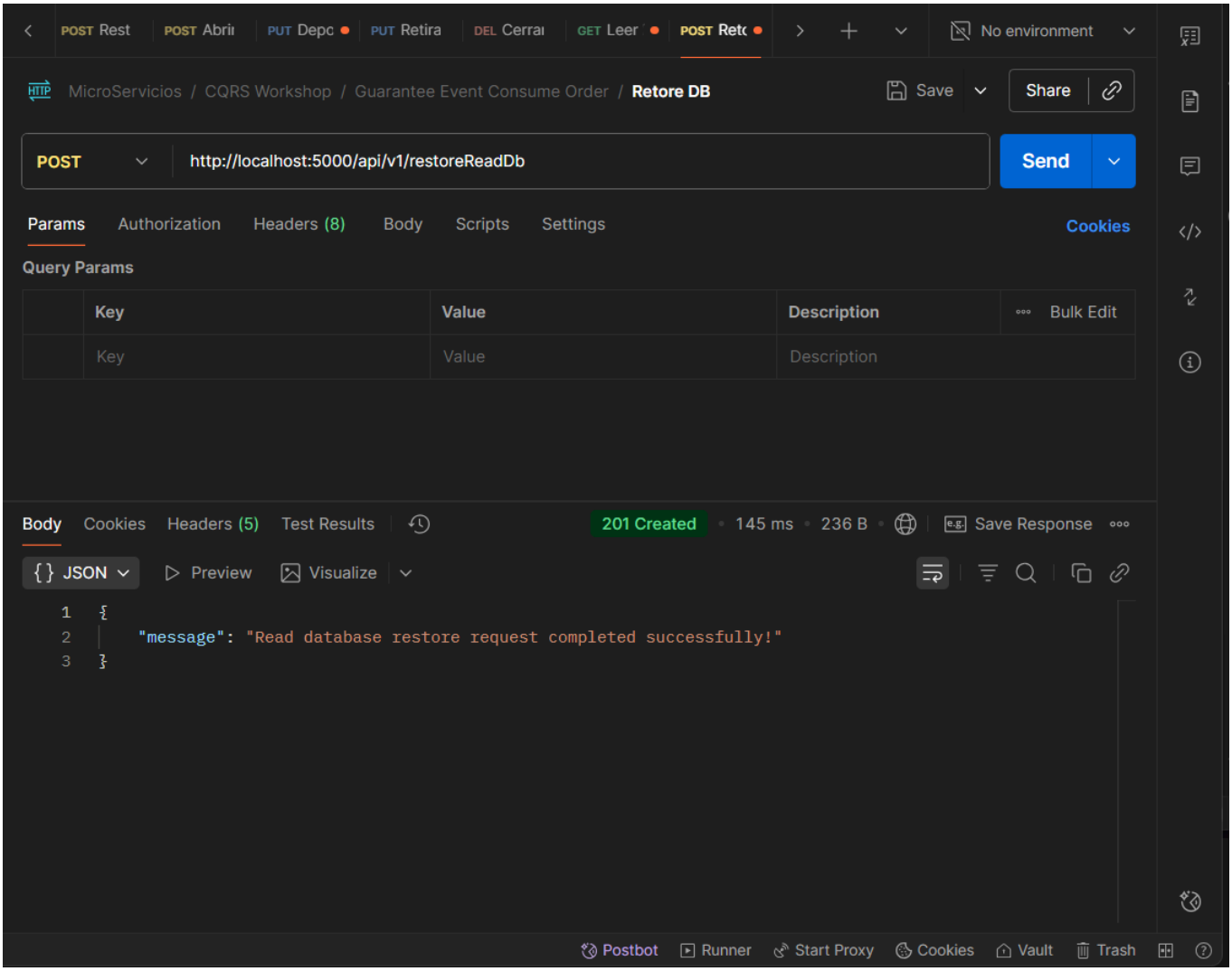


Lectura de cuenta ya cerrada

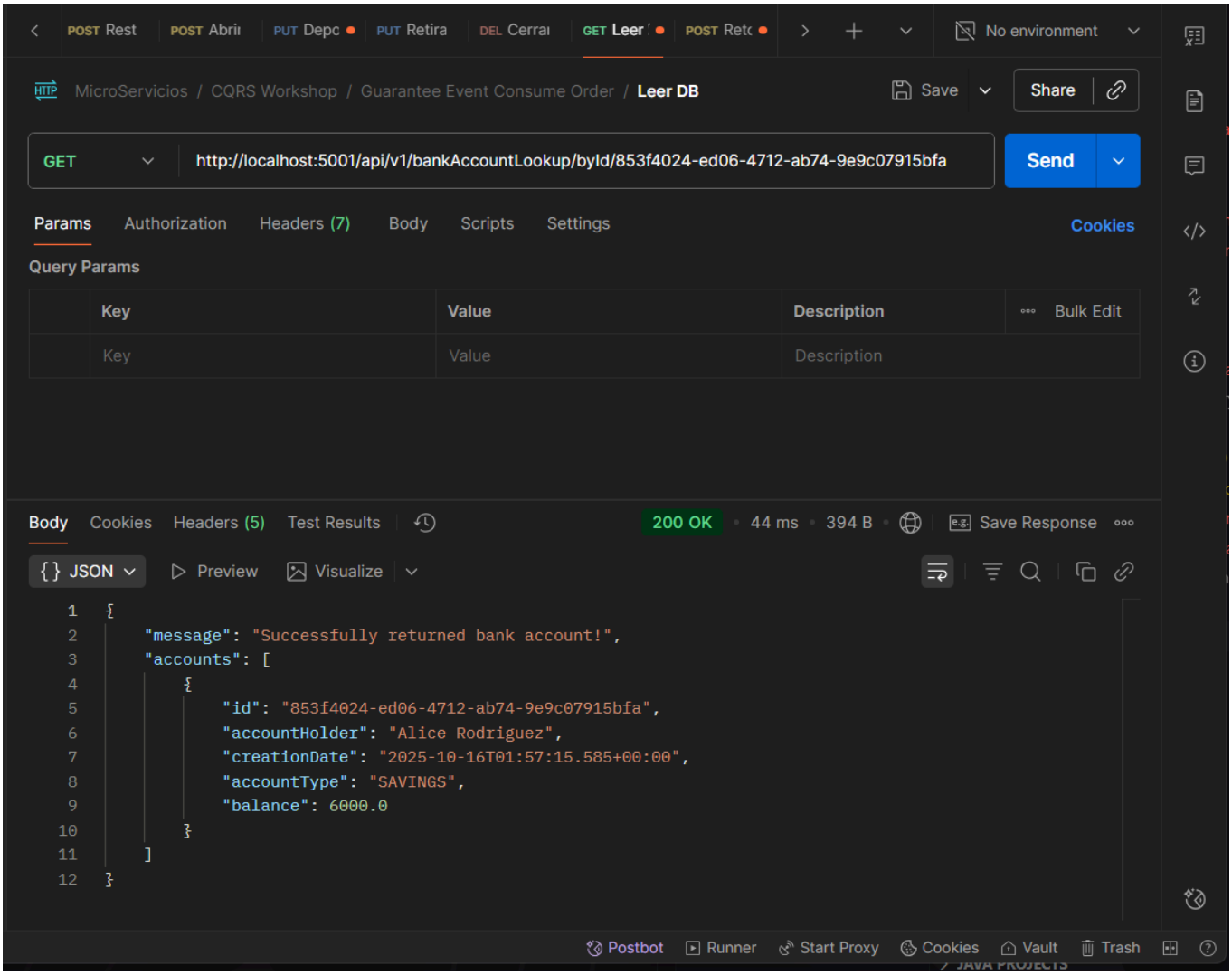

MicroServicios / CQRS Workshop / Guarantee Event Consume Order / Leer DB
Save
Share
GET
http://localhost:5001/api/v1/bankAccountLookup/byId/4655dcfd-3e3f-4561-8318-2eda150c5548
Send
Params
Authorization
Headers (7)
Body
Scripts
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Key
Value
Description
Body
Cookies
Headers (3)
Test Results
204 No Content
141 ms
112 B
Save Response
Raw
Preview
Visualize
1
Postbot
Runner
Start Proxy
Cookies
Vault
Trash

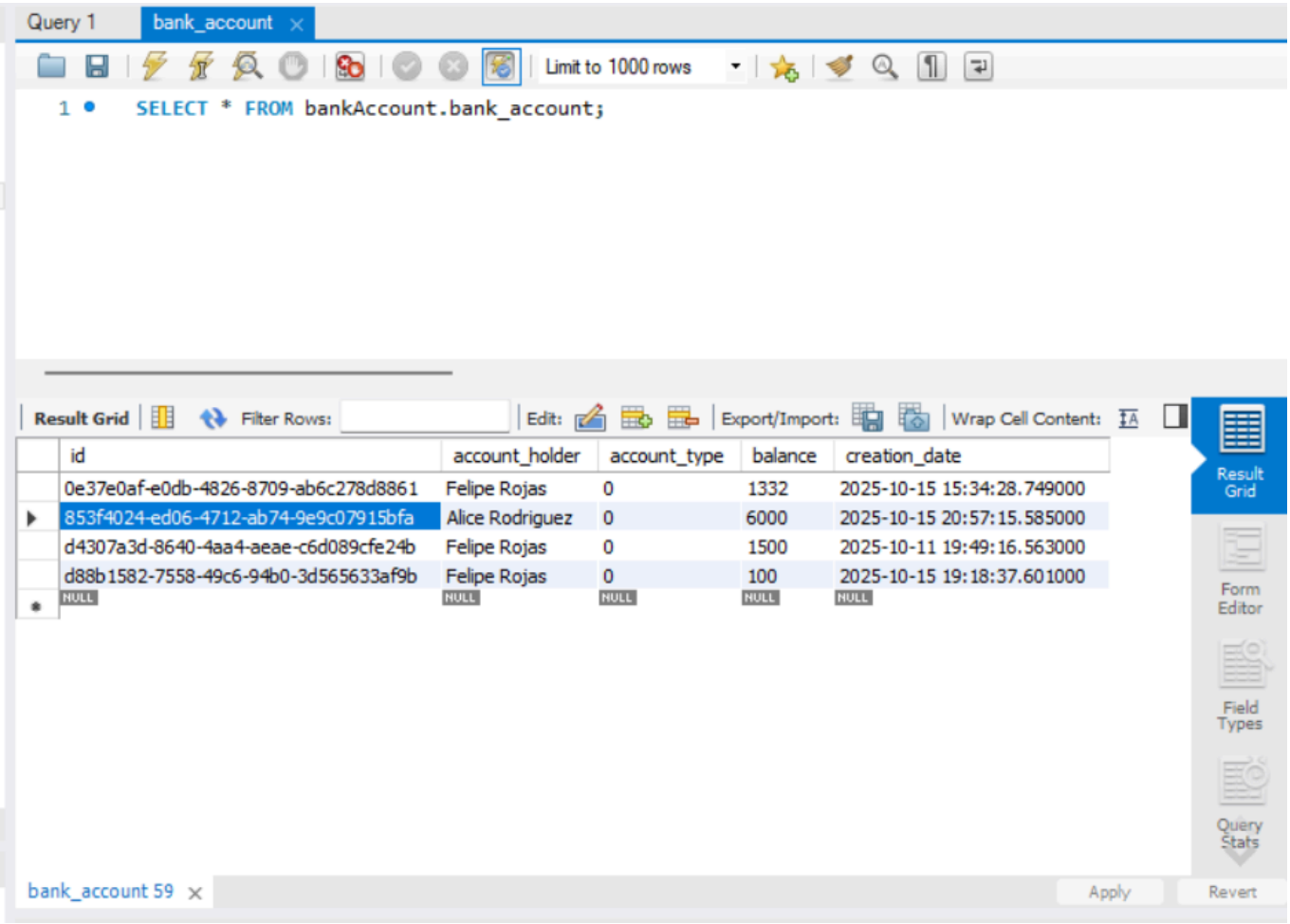
Restore DB

La cuenta fue eliminada anteriormente



Y ahora podemos ver que fue restaurada con el mensaje





Explicación de cómo se garantiza el orden de los eventos y por qué es importante en CQRS y Event Sourcing.

El orden de los eventos se garantiza asignando todas las events de una misma entidad (por ejemplo, una cuenta) a la misma «canaleta» de Kafka (partición) usando la misma key (aggregateId). Kafka mantiene el orden de los mensajes dentro de cada partición, por eso si siempre se publica con la misma key, los consumidores verán los eventos de esa entidad en el mismo orden en que fueron producidos. Por otro lado, esto es importante en CQRS + Event Sourcing porque la proyección (read model) se reconstruye aplicando los eventos en el mismo orden en que ocurrieron para obtener el mismo estado determinista de la entidad. Con lo que, ordenar = publicar con la misma key (aggregateId) para que Kafka entregue en orden dentro de la partición; es crítico para reproducir correctamente el estado de una entidad.

Resumen

Se centralizó la publicación de eventos en un único topic (`account-events`) y se garantizó el orden por entidad publicando siempre con `key = aggregateId` . Además se unificó el consumidor en el servicio de consulta para demultiplexar por un header `eventType` y se ajustó la lógica de republish/restore para conservar key y orden. Con esto la proyección (read model) puede reconstruirse determinísticamente desde el event store.

Objetivos

- Garantizar que todos los eventos de una misma entidad (aggregate) se consuman y apliquen en el mismo orden.
- Permitir reproducir (replay) todos los eventos desde el event store para reconstruir la read model.

Para Verificación:

1. Arrancar infra (Kafka, ZK, Mongo, MySQL, Kafdrop).
2. Arrancar servicios:
   - Command app (puerto 5000)
   - Query app (puerto 5001)
3. Crear cuenta (POST /api/v1/openBankAccount) — guardar `id`.
4. Hacer operaciones: PUT /api/v1/depositFunds/{id}, PUT /api/v1/withdrawFunds/{id}.
5. En Kafdrop, abrir `account-events` y comprobar:
   - Cada registro tiene `key == <aggregateId>`.
   - Header `eventType` presente.
6. Consultar read DB: GET http://localhost:5001/api/v1/bankAccountLookup/byId/{id} y verificar balances.
7. Test de replay:
   - Detener `account.query`, opcionalmente limpiar read DB, arrancar `account.query`.
   - POST http://localhost:5000/api/v1/restoreReadDb (republica eventos desde Mongo).
   - Verificar que los eventos republicados aparecen en Kafdrop y la read DB se reconstruye correctamente.

Repositorio del trabajo: https://github.com/rojas435/cqrs-event-sourcing-kafka.git