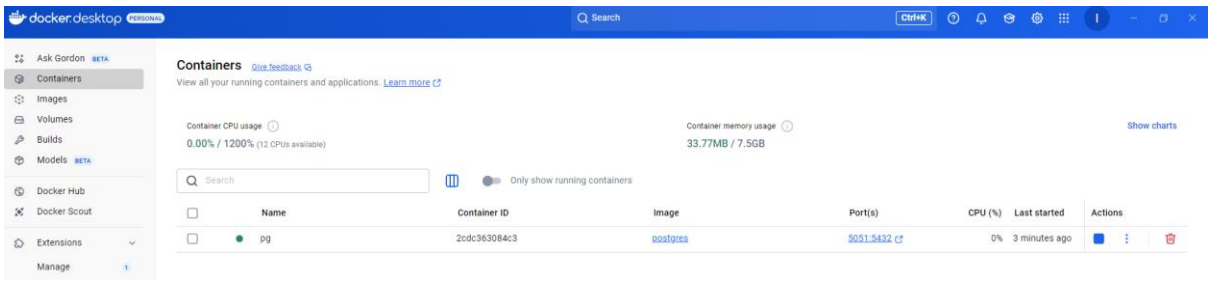
## How to setup the Back-End Java Application with Docker

### Setting up local Database

First, create a Docker container directory with the command:

docker run --name pg -e POSTGRES_USER=root -e POSTGRES_PASSWORD=Password_123 -p 5051:5432 -d postgres

```
C:\New folder\backend>docker run --name pg -e POSTGRES_USER=root -e POSTGRES_PASSWORD=Password_123 -p 5051:5432 -d postg
res
Unable to find image 'postgres:latest' locally
latest: Pulling from library/postgres
a56d6197194e: Pull complete
bbce49c39241: Pull complete
e9068395fab4: Pull complete
ebb41c3a1a26: Pull complete
67bfaf72b9da: Pull complete
3da95a905ed5: Pull complete
f68d74b3ee3b: Pull complete
761152bb4395: Pull complete
f028345d1934: Pull complete
1eb73c80cbec: Pull complete
b374722b7db6: Pull complete
c883d509c82b: Pull complete
420b84accc7f: Pull complete
9e48f0d5503b: Pull complete
Digest: sha256:3962158596daaef3682838cc8eb0e719ad1ce520f88e34596ce8d5de1b6330a1
Status: Downloaded newer image for postgres:latest
d57943ab3741f84a0f1715398a441e344e444a107418597377b733fd2883e963
```




Also from the directory where the Back-End Java project is located, use commands to see where the sql file is in the container:

docker cp demo_db.sql pg:/

docker container exec -it pg bash”

and then “ls”.

```
C:\New folder\backend>docker cp demo_db.sql pg:/
Successfully copied 2.56kB to pg:/

C:\New folder\backend>docker container exec -it pg bash
root@d57943ab3741:/# ls
bin   demo_db.sql  docker-entrypoint-initdb.d  home  lib64  mnt  proc  run   srv  tmp  var
boot  dev          etc                        lib   media  opt  root  sbin  sys  usr
root@d57943ab3741:/#
```

Then run using the command to run the commands in the SQL file in the container:

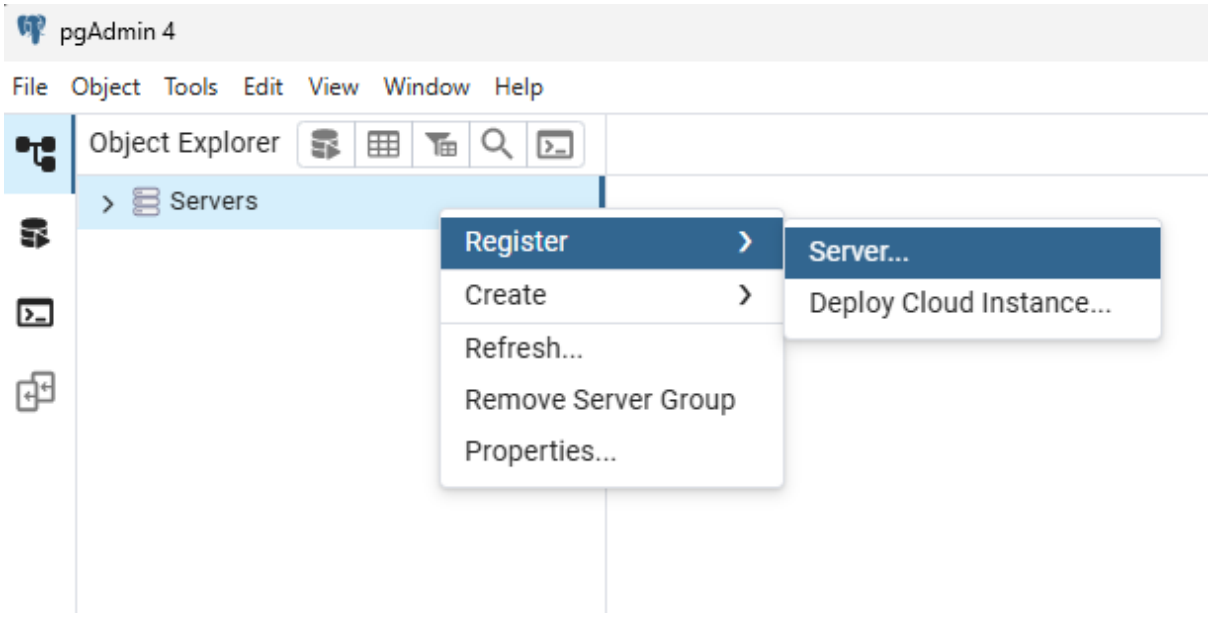
psql -U root --file demo_db.sql

demo_db.sql - not connected

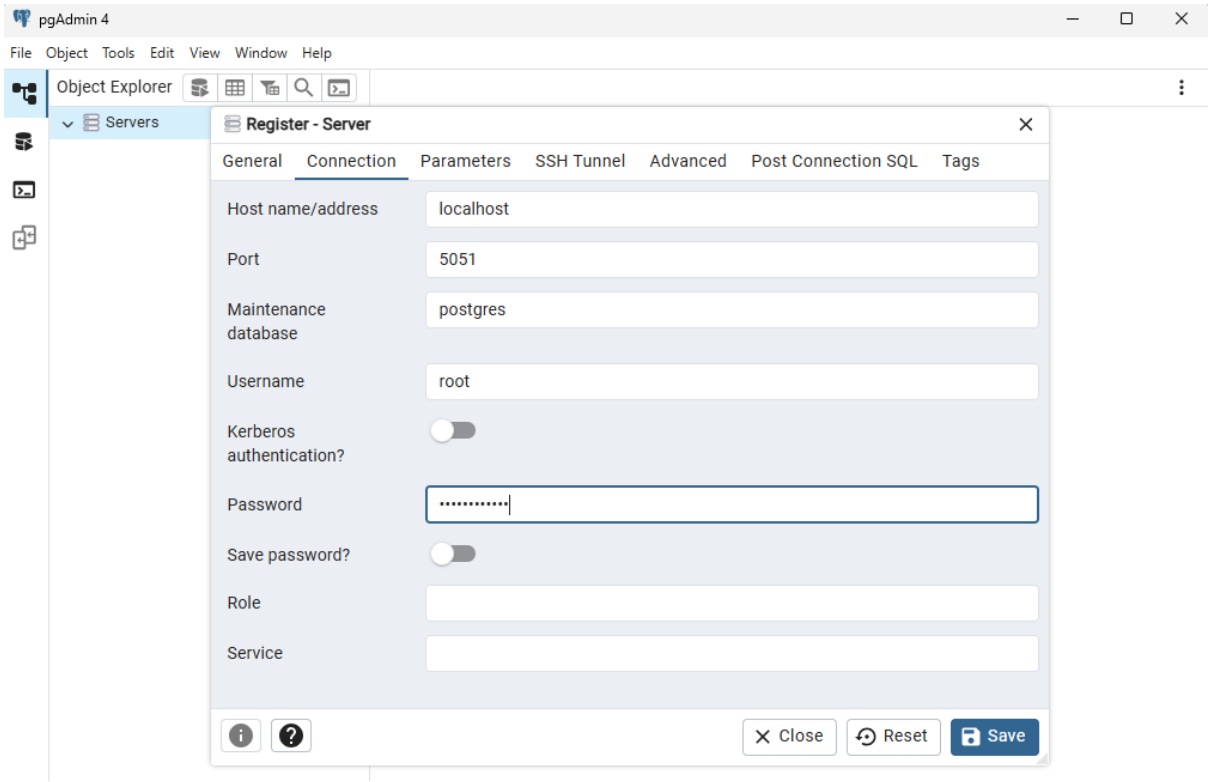
```sql
drop database demo_db;
drop user root; -- in case need to run the script again

-- Create a new user
CREATE USER root WITH PASSWORD 'Password_123';

-- Create a new database (optional)
CREATE DATABASE demo_db;
\connect demo_db

-- Grant privileges to the new user (optional)
GRANT ALL PRIVILEGES ON DATABASE demo_db TO root;

-- Create a table
CREATE TABLE demo_table (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id VARCHAR(255) NOT NULL,
    title VARCHAR(255) NOT NULL,
    amount DECIMAL(10,2) NOT NULL,
    category VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

GRANT ALL PRIVILEGES ON TABLE demo_table TO root;
```

```
root@d57943ab3741:/# psql -U root --file demo_db.sql
psql:demo_db.sql:1: ERROR:  database "demo_db" does not exist
psql:demo_db.sql:2: ERROR:  current user cannot be dropped
psql:demo_db.sql:5: ERROR:  role "root" already exists
CREATE DATABASE
You are now connected to database "demo_db" as user "root".
GRANT
CREATE TABLE
GRANT
root@d57943ab3741:/#
```

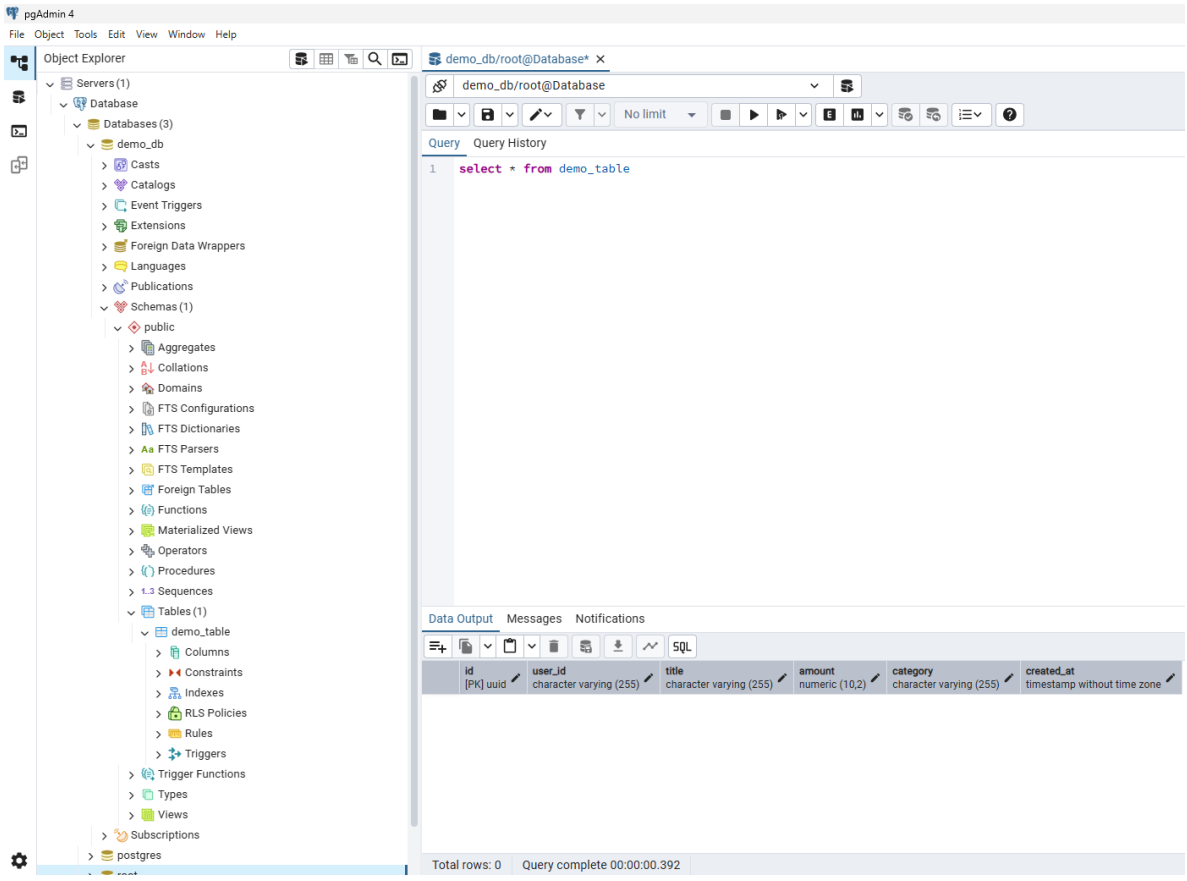
Open pgAdmin4 to register server




Register the server, under ‘General’ tab, use any name you like. Under ‘Connection’ tab input the configuration as follows, for password: Password_123.

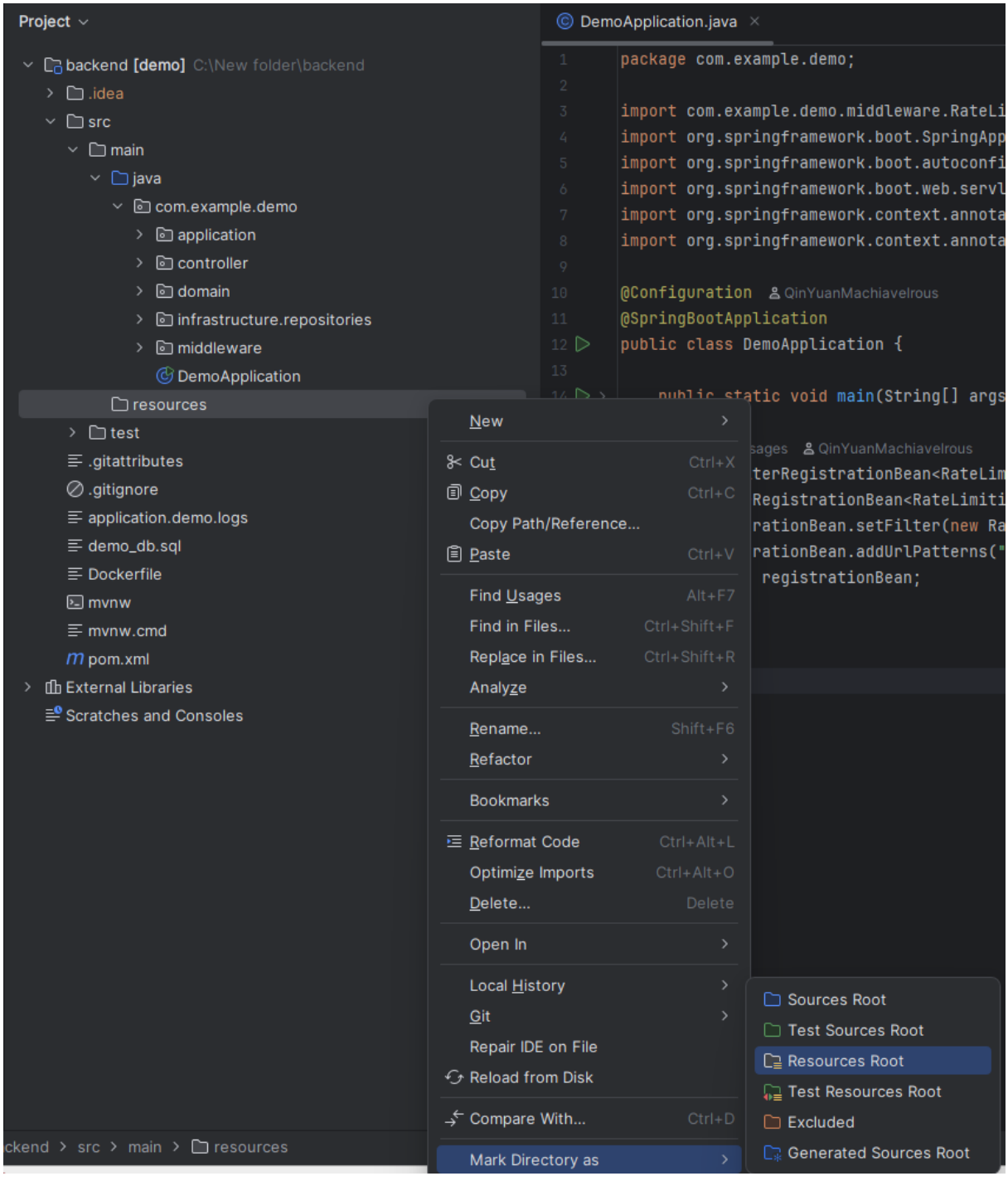



Then you will be able to access the table under *name you chose* > Databases > demo_db > Schemas > public > Tables.



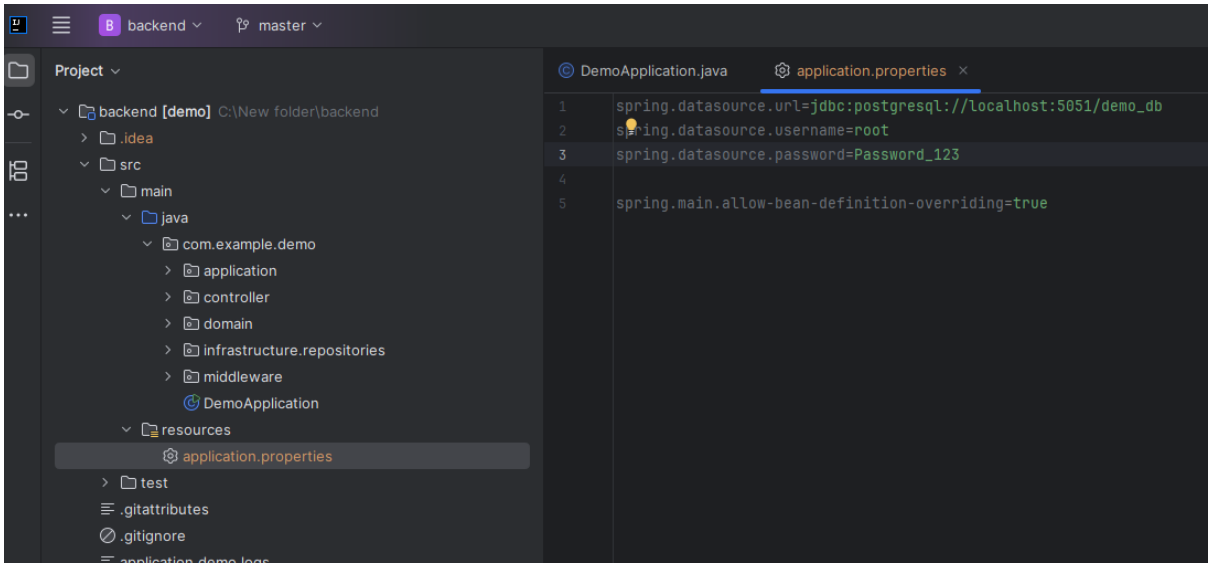
**Setting up local environment for the Back-End project**

Assuming that you are also using the IntelliJ IDEA IDE. After git pulling the project, go to src > main and create a new directory, and name it “resources” and designate it as Resources Root.

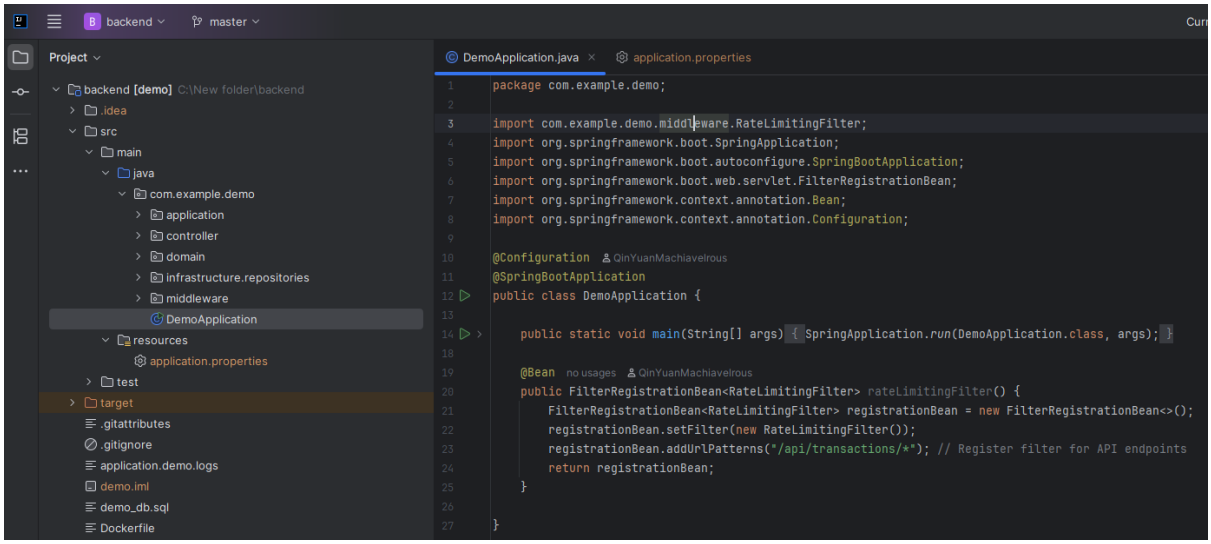



In the resources folder, create a new file named “application.properties”. This is the environment variables for the project and was thus excluded from the git repository. Paste the following lines into it:

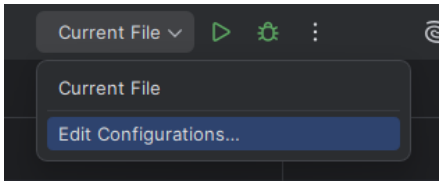
spring.datasource.url=jdbc:postgresql://localhost:5051/demo_db
spring.datasource.username=root
spring.datasource.password=Password_123
spring.main.allow-bean-definition-overriding=true



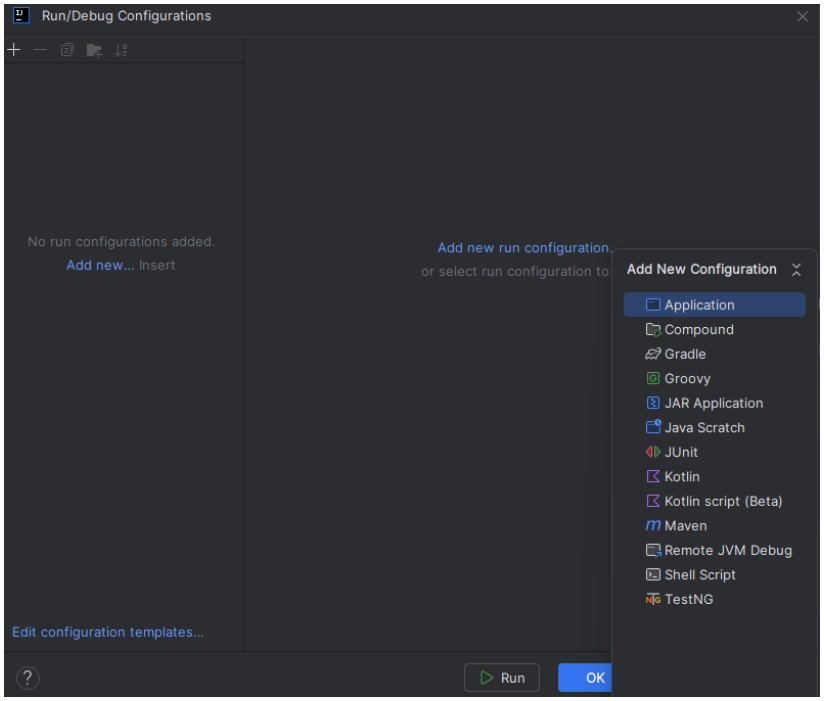

Make sure java folder is marked as Sources Root. Go to src > main > java > com.example.demo > DemoApplication.java.



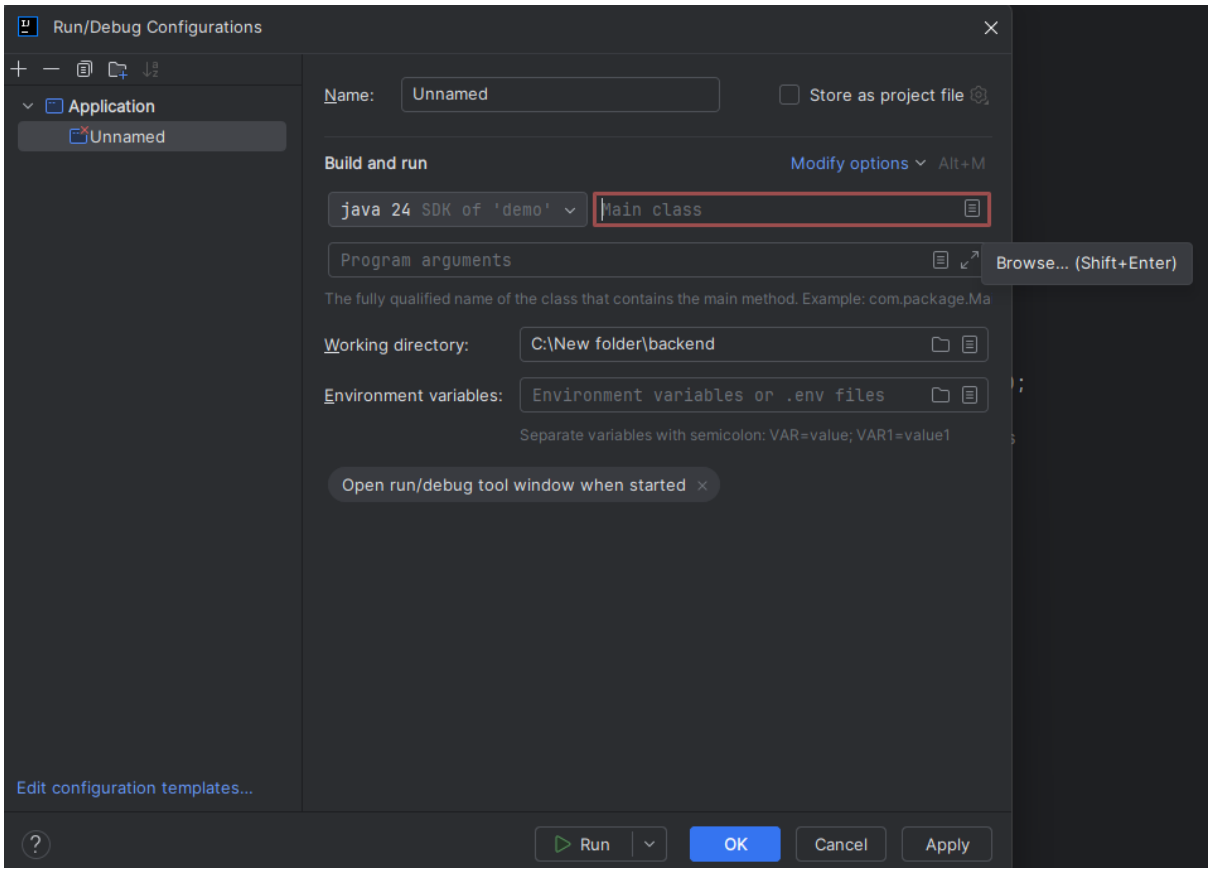

Then click Edit Configurations



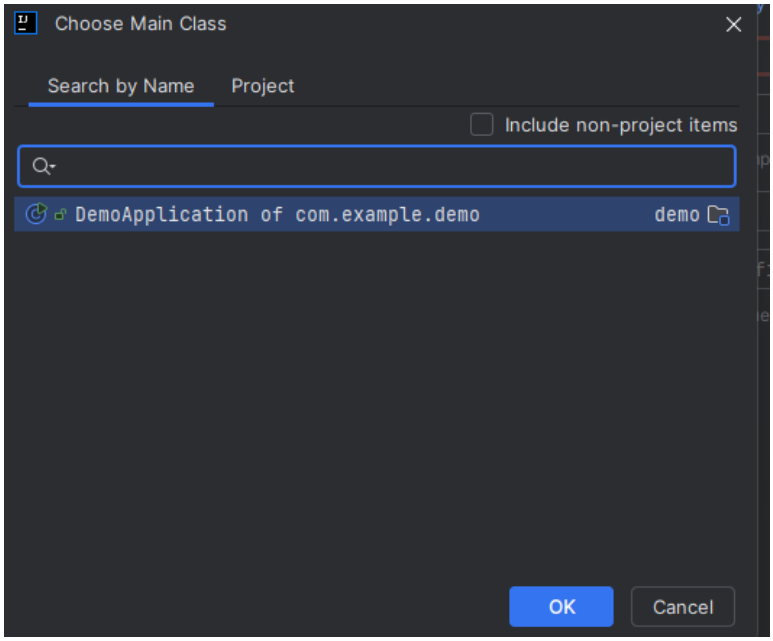

Then click Add new run configuration, and select Application.

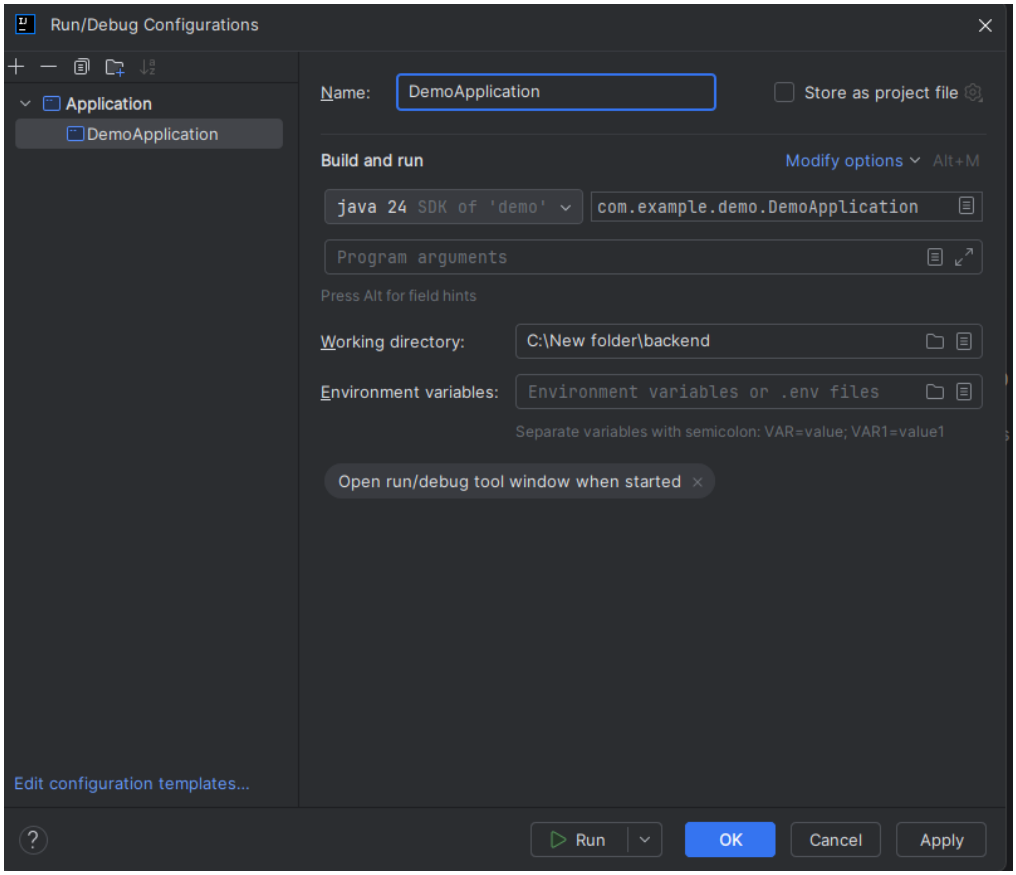



Browse for the Main class.

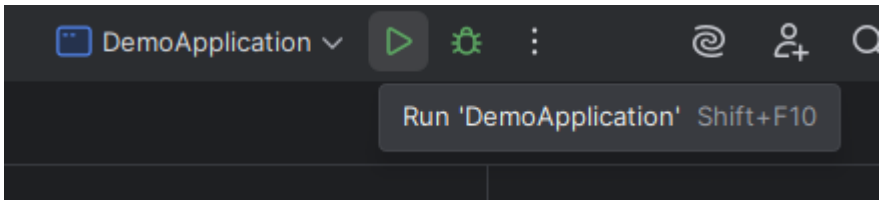



Select DemoApplication of com.example.demo. Then click “OK” and then apply and then click “OK”. Add a name to the configuration if needed.

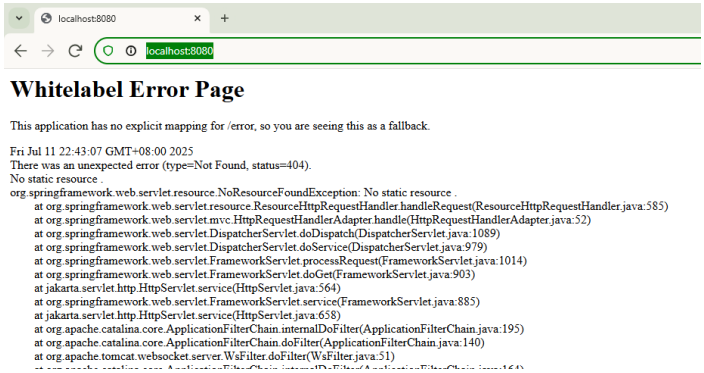



Then you can start running the Back-End project.



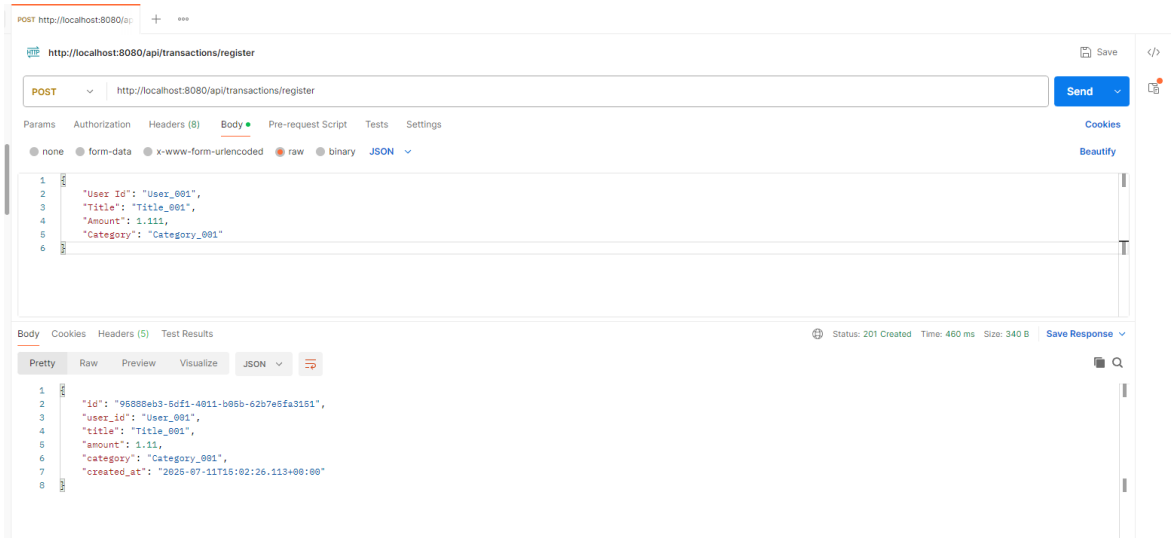
After the project starts running, try using any browser and go to http://localhost:8080/. If you get the Whitelabel Error Pager, it means the project is running successfully.

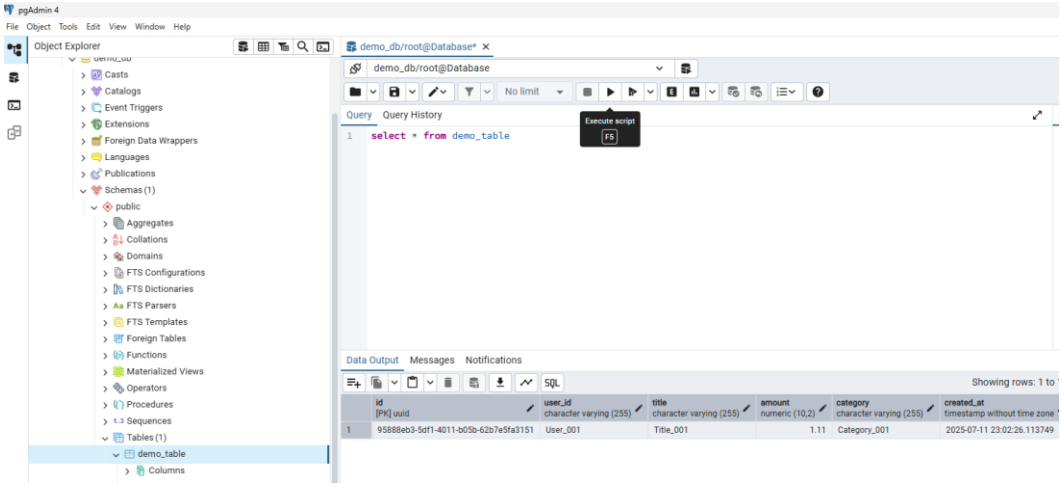



**Testing both the local database and the Back-End project.**

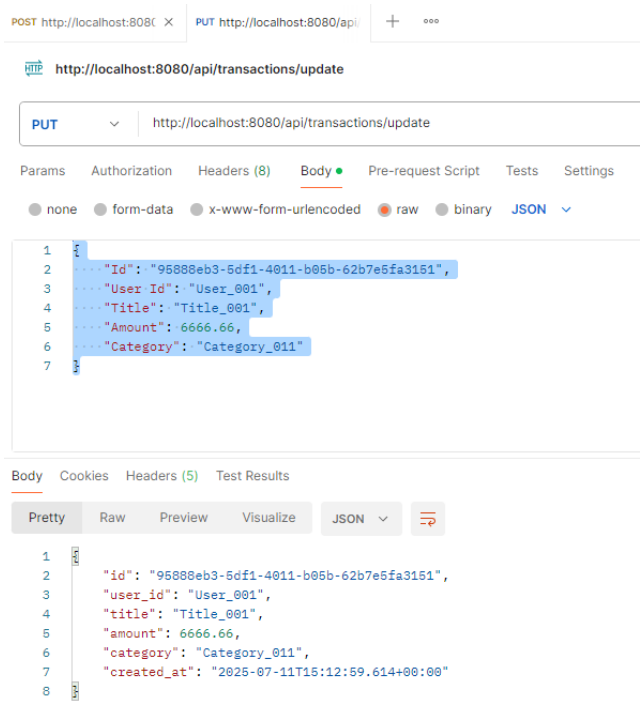
Using any API platform (using Postman for example), to test the CREATE function for the Back-End project. Use the url: http://localhost:8080/api/transactions/register and the JSON body:
{"User Id": "User_001",  "Title": "Title_001","Amount": 1.111,"Category": "Category_001"}

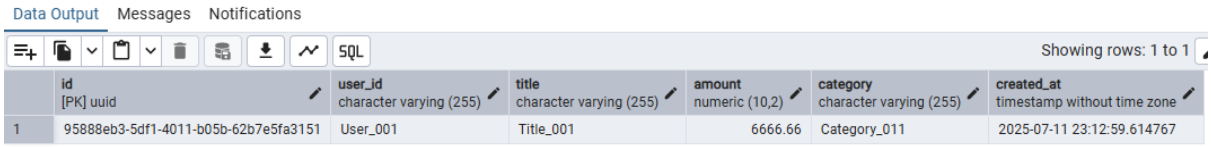
If successful, you should see the created row in the table using pgAdmin4.



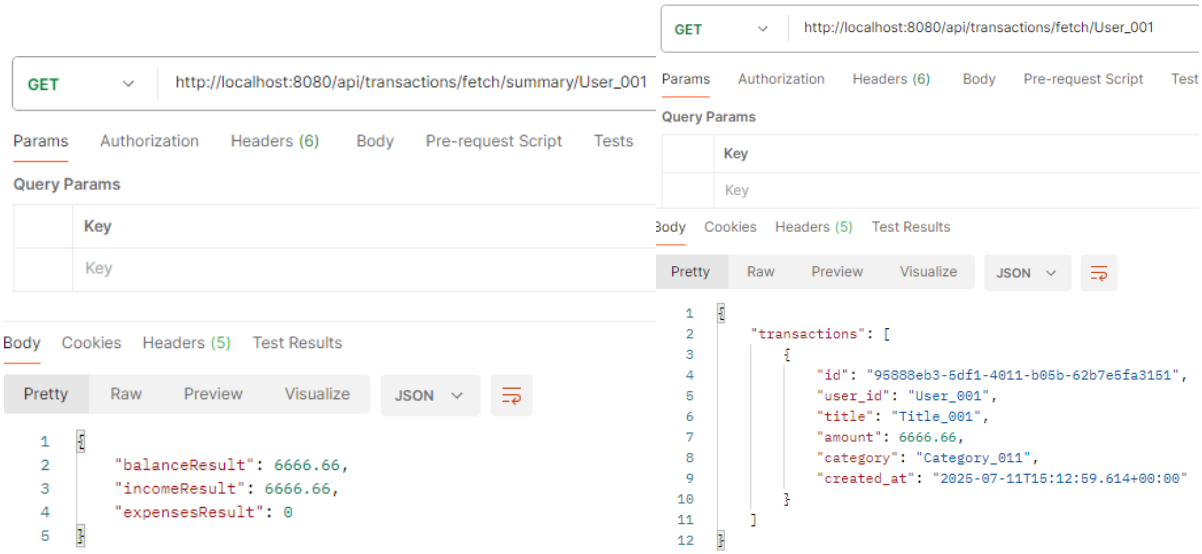
To test the UPDATE function copy the ID of the created row and use it within the body (change the fields as desired). Url: http://localhost:8080/api/transactions/update and JSON body: {"Id": *created ID*, "User Id": "User_001", "Title": "Title_001", "Amount": 6666.66, "Category": "Category_011"}.



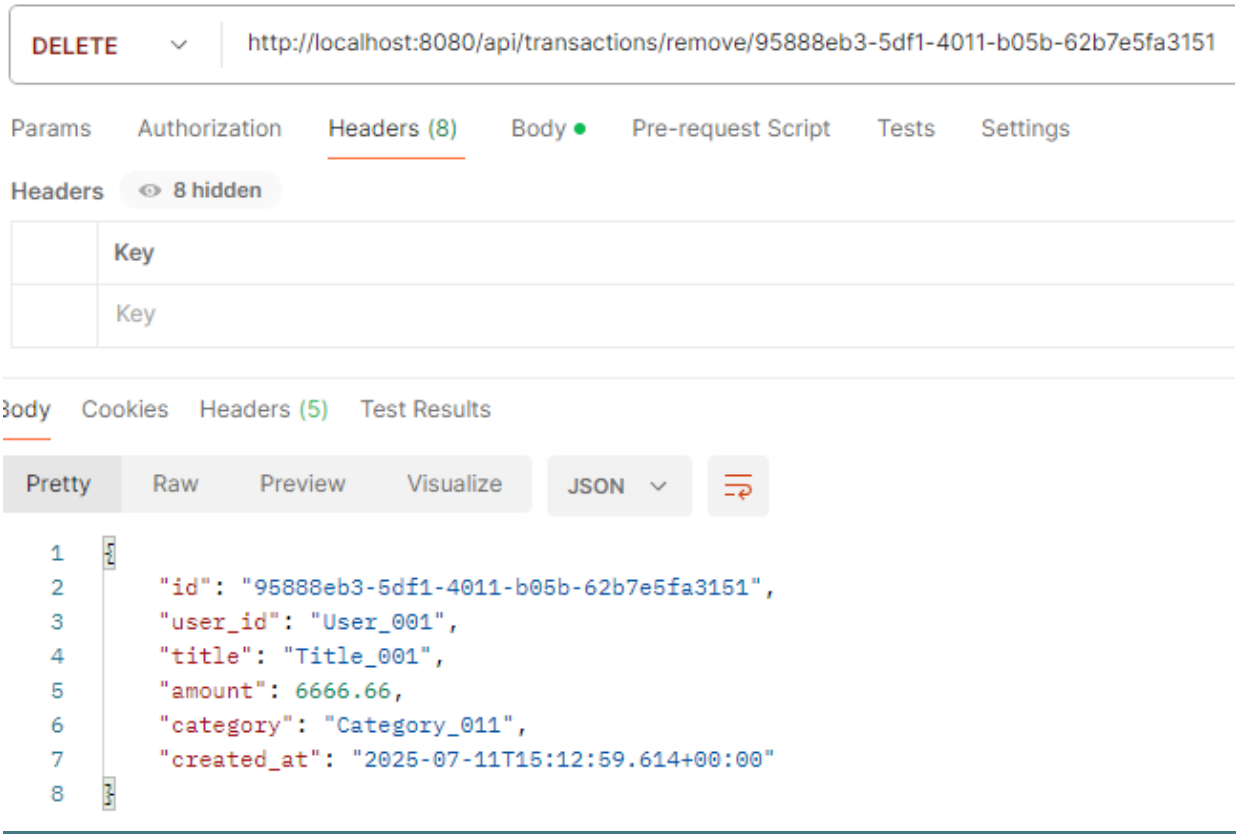



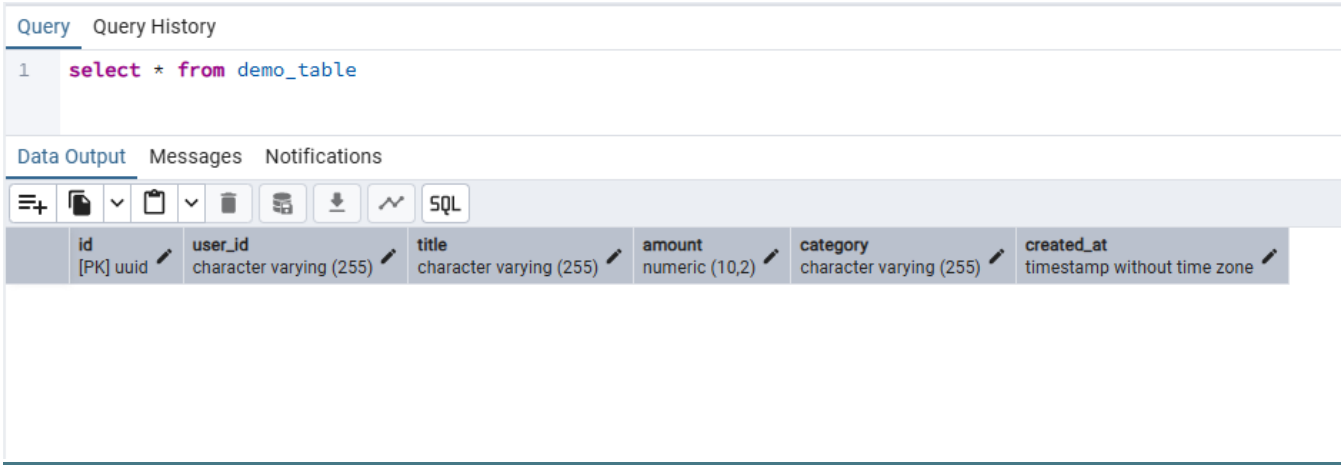

Test the READ function. Urls:
http://localhost:8080/api/transactions/fetch/summary/User_001

http://localhost:8080/api/transactions/fetch/User_001

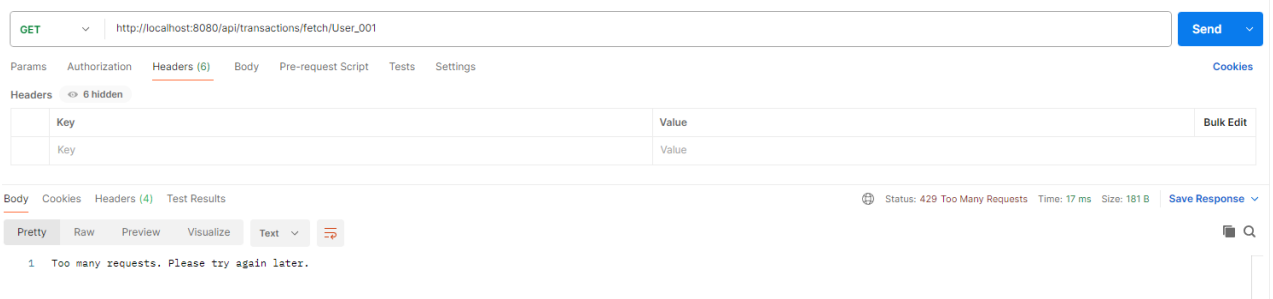


Test the DELETE function. Url: http://localhost:8080/api/transactions/remove/*ID*




Recreate a user and keep using READ function. Trigger more than 10 requests in one minute and you should see this message.



This is due to the rate limiting function implemented.

RateLimitingFilter.java | DemoApplication.java | TransactionController.java | application.properties

```java
package com.example.demo.middleware;

import jakarta.servlet.*;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;

@Component 6 usages QinYuanMachiavelrous *
public class RateLimitingFilter implements Filter {

    // Map to store request counts per IP address
    private final Map<String, AtomicInteger> requestCountsPerIpAddress = new ConcurrentHashMap<>(); 2 usages

    // Maximum requests allowed per minute
    private static final int MAX_REQUESTS_PER_MINUTE = 10; 1 usage

    @Override QinYuanMachiavelrous
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest httpServletRequest = (HttpServletRequest) request;
        HttpServletResponse httpServletResponse = (HttpServletResponse) response;

        String clientIpAddress = httpServletRequest.getRemoteAddr();

        // Initialize request count for the client IP address
        requestCountsPerIpAddress.putIfAbsent(clientIpAddress, new AtomicInteger( initialValue: 0));
        AtomicInteger requestCount = requestCountsPerIpAddress.get(clientIpAddress);

        // Increment the request count
        int requests = requestCount.incrementAndGet();

        // Check if the request limit has been exceeded
        if (requests > MAX_REQUESTS_PER_MINUTE) {
            httpServletResponse.setStatus(HttpStatus.TOO_MANY_REQUESTS.value());
            httpServletResponse.getWriter().write( s: "Too many requests. Please try again later.");
            return;
        }

        // Allow the request to proceed
        chain.doFilter(request, response);

        // Optional: Reset request counts periodically (not implemented in this simple example)
    }
}
```