PECL3

BBDD 2022/2023

Adrián Rodríguez Hurtado – DNI Oculto

Víctor Sanavia Valdeolivas – DNI Oculto

ÍNDICE

Contenido

Librerías necesarias

Todas las librerías utilizadas son propias de Python a excepción de una, "*dotenv*", la cual es necesario descargarse para que el programa en Python ***index.py***, pueda leer las contraseñas y datos de conexión de los distintos usuarios. Para ello, hay que escribir el siguiente comando en la terminal "*pip install python-dotenv*". Decidimos implementar esta librería para que las contraseñas de los distintos usuarios no estuviesen visibles y así lograr cierta privacidad y limpieza en el código del programa.

Creación de los disparadores que se necesiten para completar la lógica de negocio

Triggers de auditoría

En primer lugar, para poder almacenar las auditorias, tenemos que crear una **tabla de auditoría**.

```
CREATE TABLE peliculas.auditoria(

    evento text,
    tabla name,
    usuario text,
    fecha timestamp
);
```

Una vez creada la tabla de auditoría, creamos **la función que ejecutarán los disparador**.

```
CREATE OR REPLACE FUNCTION peliculas.fn_auditoria() RETURNS TRIGGER AS $fn_auditoria$
BEGIN
    IF current_user = 'critico' THEN
        PERFORM peliculas.da_permiso_critico_audit();
    END IF;
    IF TG_OP = 'INSERT' THEN
        INSERT INTO peliculas.auditoria(evento, tabla, usuario, fecha) VALUES ('INSERT',
TG_TABLE_NAME, current_user, current_timestamp);
    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO peliculas.auditoria(evento, tabla, usuario, fecha) VALUES ('UPDATE',
TG_TABLE_NAME, current_user, current_timestamp);
    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO peliculas.auditoria(evento, tabla, usuario, fecha) VALUES ('DELETE',
TG_TABLE_NAME, current_user, current_timestamp);
    END IF;
    IF current_user = 'critico' THEN
        PERFORM peliculas.quita_permiso_critico_audit();
    END IF;

    RETURN NEW;

END;
$fn_auditoria$ LANGUAGE plpgsql;
```

Este trigger recoge el tipo de la consulta que hemos hecho, si es un **INSERT**, introduce en la tabla de auditoría, el tipo de operación que se ha hecho, el nombre de la tabla que hizo disparar al trigger (con **TG_TABLE_NAME**), el usuario que lo ejecuto (con **current_user**) y la hora en la cual tuvo lugar dicha operación (con **current_timestamp**). Con las demás operaciones, el trigger actúa de la misma forma, variando exclusivamente el tipo de operación que añade a la tabla (**INSERT**, **UPDATE** o **DELETE**)

Como podemos ver en la imagen anterior, al principio y al final hay dos **funciones auxiliares**, las cuales hemos utilizado para solucionar el problema de que cuando el crítico insertaba una crítica, a la hora de introducir el evento en la tabla de auditoría, aparecía el usuario *postgres* en lugar de crítico. Lo que hacen estas funciones auxiliares es dar el permiso al rol de critico de poder insertar en auditoría, y cuando acaba la operación, se le quita dicho permiso, el acceso a auditoría siempre estará controlado por el disparador esto solo se ejecutará cuando el usuario sea un crítico, otro usuario (que no es crítico) ejecuta la función, esta parte no se ejecutará. Estas funciones pueden dar y quitar permisos ya que tienen un ***SECURITY DEFINER***, que lo que hace es que las funciones las ejecuta el rol o usuario que creó esas funciones, que en este caso es el usuario ***postgres***, el cual tiene todos los permisos. En primer lugar, se adjunta el código de la función que da el permiso, y después la función que quita el permiso:

```
CREATE OR REPLACE FUNCTION peliculas.da_permiso_critico_audit() RETURNS
void SECURITY DEFINER AS $BODY$
BEGIN
    GRANT INSERT ON peliculas.auditoria TO critico;
END;
$BODY$
LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION peliculas.quita_permiso_critico_audit()
RETURNS void SECURITY DEFINER AS $BODY$
BEGIN
    REVOKE INSERT ON peliculas.auditoria FROM critico;
END;
$BODY$
LANGUAGE plpgsql;
```

También hacemos uso de las variables **current_timestamp** (que devuelve un timestamp con la fecha, día y hora actual) y del **current_user** (que devuelve el nombre del usuario que está activo en ese momento) además de las operaciones de los triggers o disparadores que encontramos en la **documentación oficial de PostgreSQL** como son el **TG_OP** (que devuelve el tipo de operación que ha disparado el trigger) y **TG_TABLE_NAME** (que devuelve el nombre de la tabla que invocó al trigger), cuyas documentaciones oficiales están aquí.

Una vez que hemos creado todas las funciones necesarias para que el disparador de auditoría funcione correctamente, **creamos los disparadores de audotoria de todas las tablas**:

Trigger de auditoria para la tabla *peliculas.crticas*:

```
CREATE TRIGGER tg_criticas_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.criticas
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.personal*:

```
CREATE TRIGGER tg_personal_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.personal
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.actor*:

```
CREATE TRIGGER tg_actor_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.actor
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.director*:

```
CREATE TRIGGER tg_director_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.director
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.actua*:

```
CREATE TRIGGER tg_actua_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.actua
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.peliculas*:

```
CREATE TRIGGER tg_peliculas_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.peliculas
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.generos*:

```
CREATE TRIGGER tg_generos_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.generos
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.caratulas*:

```
CREATE TRIGGER tg_caratulas_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.caratulas
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.pag_web*:

```
CREATE TRIGGER tg_paginaweb_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.pag_web
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Este disparador se encargará de almacenar dentro de la tabla ***películas.auditoria*** los eventos que van teniendo lugar en las distintas tablas de la base de datos, los cuales tienen esta información / atributos:

- La tabla en la que ha ocurrido el evento.
- El tipo de evento u operación.
- El usuario que lo ha llevado a cabo.
- La fecha y la hora en la cual tuvo lugar la inserción.

Trigger de insertar críticas

Este trigger está relacionado con las críticas. Debe comprobar si, al insertar una crítica, la página web se encuentra o no en la tabla. Si ésta no se encuentra, se añadirá a la tabla.

En primer lugar, **creamos la función que ejecutará el trigger de insertar críticas**. Cuyo funcionamiento se explica a continuación

Primero, vamos a comprobar que el usuario ha introducido todos los atributos necesarios, es decir, los que en la tabla de ***peliculas.criticas*** son ***NOTNULL***.

```
CREATE OR REPLACE FUNCTION peliculas.fn_inserta_critica() RETURNS TRIGGER
SECURITY DEFINER AS $fn_inserta_critica$

BEGIN

    IF NEW.critico ISNULL THEN
        RAISE EXCEPTION 'El nombre del critico no puede ser nulo, debes
imprimir uno';
    END IF;

    IF NEW.anno_peliculas ISNULL THEN
        RAISE EXCEPTION 'El anno de la pelicula no puede ser nulo, debes
aportar uno';
    END IF;

    IF NEW.titulo_peliculas ISNULL THEN
        RAISE EXCEPTION 'El titulo de la pelicula no puede tener valor
nulo, debes aportar uno';
    END IF;

    IF NEW.nombre_pag_web ISNULL THEN
        RAISE EXCEPTION 'El nombre o url de la pagina web no puede ser
nulo, debes aportar uno';
    END IF;
```

Si el usuario no ha introducido todos los datos, el propio trigger producirá un error avisando al usuario de que falta ese atributo en concreto.

Si el usuario ha introducido todos los datos, se procederá a comprobar si la página web se encuentra o no en la tabla:

```
IF NEW.nombre_pag_web NOT IN (SELECT nombre FROM peliculas.pag_web) THEN
        INSERT INTO peliculas.pag_web(nombre) VALUES
(NEW.nombre_pag_web);

    END IF;

    RETURN NEW;

END;

$fn_inserta_critica$ LANGUAGE plpgsql;
```

Una vez creada la función a ejecutar, **creamos el trigger *tg_inserta_critica***, que hará uso de la función anteriormente descrita:

```
CREATE TRIGGER tg_inserta_critica

    BEFORE INSERT
    ON peliculas.criticas
    FOR EACH ROW
    EXECUTE FUNCTION peliculas.fn_inserta_critica();
```

Para solucionar el error de que al insertar critica, en auditoria aparecía siempre el usuario ***postgres*** hicimos lo mismo que en auditoria, es decir, dar permisos a critico (en caso de que el trigger sea llamado por este usuario) para insertar en ***películas.pag_web*** y una vez que ya se ha insertado la pagina web, se le quita el permiso, de este modo, **el acceso a este privilegio está controlado por este trigger**. Las funciones que usamos para conseguir esto están a continuación: (en primer lugar, tendremos la que da el permiso, y después tenemos la que quita el permiso)

```
CREATE OR REPLACE FUNCTION peliculas.da_permiso_critico_pagweb() RETURNS
void SECURITY DEFINER AS $BODY$
BEGIN
    GRANT INSERT ON peliculas.pag_web TO critico;
END;
$BODY$
LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION peliculas.quita_permiso_critico_pagweb()
RETURNS void SECURITY DEFINER AS $BODY$
BEGIN
    REVOKE INSERT ON peliculas.pag_web FROM critico;
END;
$BODY$
LANGUAGE plpgsql;
```

Trigger de media

Para este tercer y último trigger se creará una nueva tabla para almacenar las medias de las películas. La tabla deberá de ser actualizada cada vez que se introduzca una nueva crítica.

La tabla donde se almacenarán las medias será ***películas.nota_media_peliculas***:

```
CREATE TABLE peliculas.nota_media_peliculas(

    titulo_peliculas text,
    anno_peliculas smallint,
    media integer
);
```

Para un correcto funcionamiento de este trigger, creamos una vista que contiene todas las medias de las películas actualizadas cada vez que la ejecutamos (este código es reutilizado de nuestra tercera parte de la práctica 2):

```
CREATE VIEW peliculas.media_peliculas as
(SELECT titulo_peliculas, anno_peliculas, avg(puntuacion) as
puntuacion_media
FROM
peliculas.criticas
GROUP BY
titulo_peliculas, anno_peliculas
ORDER BY
avg(puntuacion));
```

Este trigger se activará después de una inserción dentro de la tabla ***películas.criticas***, y actualizará la media de la película de la que se acaba de insertar la crítica, ya que actualizar las demás no tiene sentido, ya que seguirán teniendo la misma media.

```
CREATE OR REPLACE FUNCTION peliculas.fn_actualiza_media_peliculas()
RETURNS TRIGGER SECURITY DEFINER AS $fn_actualiza_media_peliculas$

BEGIN

    UPDATE peliculas.nota_media_peliculas SET (media) =
    (SELECT puntuacion_media
    FROM peliculas.media_peliculas
    WHERE (peliculas.media_peliculas.titulo_peliculas =
NEW.titulo_peliculas) and (peliculas.media_peliculas.anno_peliculas =
NEW.anno_peliculas));

    RETURN NEW;

END;

$fn_actualiza_media_peliculas$ LANGUAGE plpgsql;
```

Se crea el trigger ***tg_actualiza_medias_peliculas*** que se disparará después de una inserción dentro de la tabla ***películas.criticas*** para **actualizar la media de las películas**:

```
CREATE TRIGGER tg_actualiza_medias_peliculas
    AFTER INSERT
    ON peliculas.criticas
    FOR EACH ROW
    EXECUTE FUNCTION peliculas.fn_actualiza_media_peliculas();
```

Utilizamos esta consulta auxiliar para poblar la tabla de las medias basándonos en nuestra vista anteriormente creada:

```
INSERT INTO peliculas.nota_media_peliculas(titulo_peliculas,
anno_peliculas, media)
SELECT titulo_peliculas, anno_peliculas, puntuacion_media FROM
peliculas.media_peliculas;
```

Creación de usuarios

El programa constará de 4 tipos distintos de usuarios:

- **Administrador** → Todos los permisos.
- **Gestor** → Inserción, modificación, borrado y consulta.
- **Crítico** → Consulta de todas las tablas a excepción de la tabla "auditoria" e inserción en la tabla "críticas".
- **Cliente** → Consulta de todas las tablas a excepción de la tabla "auditoria".

Primero, crearemos los usuarios con sus respectivas contraseñas.

```
CREATE USER admin PASSWORD 'admin';
CREATE USER gestor PASSWORD 'gestor';
CREATE USER critico PASSWORD 'critico';
CREATE USER cliente PASSWORD 'cliente';
```

Una vez creados, les asignamos los permisos en función del rol.

```
REVOKE ALL PRIVILEGES ON ALL TABLES IN SCHEMA peliculas FROM gestor,
critico, cliente;
GRANT USAGE ON SCHEMA peliculas TO admin, gestor, critico, cliente;
GRANT SELECT ON ALL TABLES IN SCHEMA peliculas TO critico;
GRANT INSERT ON peliculas.criticas TO critico;
GRANT SELECT ON ALL TABLES IN SCHEMA peliculas TO cliente;

CREATE TABLE peliculas.auditoria(

    evento text,
    tabla name,
    usuario text,
    fecha timestamp
);

GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA peliculas TO admin WITH
GRANT OPTION;
GRANT INSERT, UPDATE, DELETE, SELECT ON ALL TABLES IN SCHEMA peliculas TO
gestor;
```

Como se muestra en el código, en primer lugar, hacemos uso de la función ***REVOKE*** para eliminar todos los permisos de todos los roles que puedan tener predeterminados. Acto seguido, damos un ***GRANT USAGE*** a todos los roles para que puedan tener acceso al esquema ***películas***. Y después damos permisos de ***SELECT*** e ***INSERT*** en todas las tablas del esquema ***películas*** al usuario *crítico* y el permiso ***SELECT*** en todas las tablas del esquema ***películas*** a ***cliente***. Nótese que hemos creado la **tabla de auditoría** justo después de dar estos permisos, porque hemos considerado que la información de auditoría solo la podrían ver los usuarios ***admin*** y ***gestor*** ya que es información que no le interesa saber a un ***cliente*** o a un ***crítico***, y así, dicha tabla es invisible tanto para ***crítico*** como para ***cliente***.

Después de crear la tabla de auditoría damos todos los permisos en el esquema ***películas*** a ***admin*** con la opción de dar permisos. Y, por último, damos los permisos de *INSERT*, ***UPDATE***, ***DELETE*** y ***SELECT*** en todas las tablas del esquema *películas* a ***gestor***.

Aquí adjuntamos un link para ver la documentación oficial de la sentencia ***GRANT*** y de los permisos que ofrece.

Conexión con programas externos y seguridad

Se ha creado un programa en Python para que cualquier usuario creado previamente pueda conectarse a la base de datos. El programa solicitará el usuario, su respectiva contraseña y la consulta a resolver o la crítica a insertar.

Primero, se llevará a cabo la ***elección del usuario***, donde el usuario tendrá que elegir uno de los 4 usuarios creados previamente, e introducir la contraseña asignada a cada uno. Si la contraseña es correcta, el programa mostrará el texto 'Contrasena correcta!'. Si ésta fuese incorrecta, no se podría asignar el usuario correspondiente. En el caso de que se hayan introducido valores que no sean números entre 1 y 4, no se podrá avanzar en la ejecución del programa, ya que todo el código de esta función esta controlado mediante un bloque ***try – except***. Si todo ha ido bien, se devolverá la información del usuario para conectarse a la base de datos, la cual se recoge en el archivo ***.env*** mediante el uso de la librería ***dotenv***.

```
def user_choice() -> str:

    global user
    correct_choice = False

    while (not correct_choice):
        try:
            print('\t\t------------------=[ELECCION DE USUARIO]=----------
-------\n\n\t1. Admin\n\t2. Gestor\n\t3. Critico\n\t4. Cliente\n\n\t\t---
--------------=[ELECCION DE USUARIO]=-----------------\n\n')
            choice = int(input('Elige el usuario con el que deseas
conectarte a la base de datos (numero): '))
            if (choice.__eq__(1)):
                password = str(input('Introduce la contrasena del usuario
admin: '))
                if (password.__eq__(os.getenv('admin_password'))):
                    user = 'admin'
                    correct_choice = True
                else:
                    print('Contrasena incorrecta para el usuario admin')
                    time.sleep(2)
            elif (choice.__eq__(2)):
                password = str(input('Introduce la contrasena del usuario
gestor: '))
                if (password.__eq__(os.getenv('gestor_password'))):
                    user = 'gestor'
                    correct_choice = True
                else:
```

```
                    print('Contrasena incorrecta para el usuario gestor')
                    time.sleep(2)
            elif (choice.__eq__(3)):
                password = str(input('Introduce la contrasena del usuario
critico: '))
                if (password.__eq__(os.getenv('critico_password'))):
                    user = 'critico'
                    correct_choice = True
                else:
                    print('Contrasena incorrecta para el usuario
critico')
                    time.sleep(2)
            elif (choice.__eq__(4)):
                password = str(input('Introduce la contrasena del usuario
cliente: '))
                if (password.__eq__(os.getenv('cliente_password'))):
                    user = 'cliente'
                    correct_choice = True
                else:
                    print('Contrasena incorrecta para el usuario
cliente')
                    time.sleep(2)
            else:
                print('Introduce una opcion correcta (numero del 1 al
4)')
                time.sleep(2)
        except ValueError:
            print('Introduce una opcion valida (numero del 1 al 4)')

        os.system('cls')

    print('Contrasena correcta!, estableciendo conexion...')
    time.sleep(2)
    user_info = os.getenv(user + '_user')

    return user_info
```

Se crearán dos funciones para **establecer conexión** con la base de datos y otra para **cerrar la conexión**. Para establecer la conexión, se hace uso de la función ***connection_establishment()***, que se encarga de conectarse mediante la orden ***psycopg2.connect()*** pasando como parámetro la información del usuario con el que se desea conectarse (recogida en el archivo ***.env*** mediante la librería ***dotenv***) y una vez conectado a la base de datos creamos el cursor mediante la orden ***connection.cursor()***.

En la función de cerrar conexión (***connection_termination()***) simplemente se cierra en primer lugar el cursor mediante la orden ***cursor.close()*** y la conexión mediante la orden ***connection.close()***.

```
def connection_establishment(user_info: str):

    global connection
    global cursor
    connection = psycopg2.connect(user_info)
    cursor = connection.cursor()
    print('\nConexion establecida correctamente!\n')
    time.sleep(1)
```

```
def connection_termination():

    global connection
    global cursor
    cursor.close()
    connection.close()
    print('Conexion cerrada.')
```

Se crea una función ***select_query()*** que recibe como argumento la consulta SQL a ejecutar. Esta función tiene el objetivo de ejecutar la consulta SQL mediante la orden ***cursor.execute()*** que recibe como parámetro dicha consulta. Una vez ejecutada, almacenaremos en la variable ***rows*** todas las filas de dicha consulta, que nos las devuelve la orden ***cursor.fetchall()***. Una vez almacenadas todas las filas, se imprimirá un mensaje por pantalla diciendo que toda ha ido bien y 1 segundo después (mediante la orden ***time.sleep(1)***) se mostrarán todos los resultados ejecutando la sentencia ***for*** que recorrerá todas las filas, mostrándolas todas por pantalla. En el caso de que haya algún error se nos informará de ello, y se nos dará información adicional sobre él, ya que todo lo mencionado anteriormente está controlado mediante un bloque ***try – except***.

```
def select_query(sql_command: str):

    global connection
    global cursor

    try:
        cursor.execute(sql_command)
        rows = cursor.fetchall()
        print('\n\nConsulta realizada correctamente, mostrando
resultados...\n\n')
        time.sleep(1)
        for row in rows:
            print(row)

    except (errors.UndefinedTable) as undefined_table:
        print(f'\n\nLa tabla que has introducido no existe ->
{undefined_table}')
    except (errors.UndefinedColumn) as undefined_column:
        print(f'\n\nLa columna introducida no existe ->
{undefined_column}')
    except (errors.InsufficientPrivilege) as permission_error:
        print(f'\n\nEl usuario elegido ({user}) no tiene permisos para
realizar esta accion -> {permission_error}')
    except (errors.SyntaxError) as syntax_error:
        print(f'\n\nError en la sintaxis de la consulta SQL ->
{syntax_error}')
```

Se crea una función ***insert_query()*** que recibe como argumento la consulta SQL a ejecutar. Esta función tiene como objetivo ejecutar la consulta SQL (que, en este caso, es de tipo insert) por el usuario. Primero ejecutaremos la consulta mediante la orden ***cursor.execute()***. Una vez que se haya ejecutado la consulta, tendremos que guardar los cambios realizados mediante la orden ***connection.commit()***. En el caso de que haya un error en alguna parte de la consulta, se nos avisará de ello y se nos dará información adicional sobre él, ya que todo el proceso anteriormente mencionado está controlado mediante un bloque ***try – except***.

```
def insert_query(sql_command: str):

    global connection
    global cursor

    try:
        cursor.execute(sql_command)
        connection.commit()
        print('\nConsulta realizada correctamente!')
    except (errors.UndefinedTable) as undefined_table:
        print(f'\n\nLa tabla que has introducido no existe ->
{undefined_table}')
    except (errors.UndefinedColumn) as undefined_column:
        print(f'\n\nLa columna introducida no existe ->
{undefined_column}')
    except (errors.InsufficientPrivilege) as permission_error:
        print(f'\n\nEl usuario elegido ({user}) no tiene permisos para
realizar esta accion -> {permission_error}')
    except (errors.UniqueViolation) as unique_violation:
        print(f'\n\nLa consulta viola una restriccion de unicidad (llave
ya existente) -> {unique_violation}')
    except (errors.ForeignKeyViolation) as foreign_key_violation:
        print(f'\n\nViolacion de clave foranea (no esta presente en la
tabla) -> {foreign_key_violation}')
    except (errors.SyntaxError) as syntax_error:
        print(f'\n\nError en la sintaxis de la consulta SQL ->
{syntax_error}')
```

Se crea la función ***yes_or_no_choice()***, que tiene como objetivo que el usuario elija entre si o no una decisión en concreto. Devuelve ***True*** en el caso de que el usuario elija ***sí***, y devolverá ***False*** cuando se elija un ***no***. En el caso de que se inserte una opción no valida, es decir que no sea ni **si** ni **no** el programa ignorará lo que se haya insertado hasta que se introduzca si o no, ya que todo el código de esta función esta controlado por un bloque ***try – except***.

```
def yes_or_no_choice() -> bool:
    result: bool
    choice_succeded = False
    choice = ''
    while (not choice_succeded):
        try:
            choice = str(input(''))
            if (choice.__eq__('si')):
                result = True
                choice_succeded = True
            elif (choice.__eq__('no')):
                result = False
                choice_succeded = True
        except ValueError:
            print('\n\nIntroduce una opcion valida (si / no)')

    return result
```

Por último, creamos la función ***main()***, que representa el hilo principal o main del programa. En primer lugar se limpia la pantalla mediante la orden ***os.system('cls')***, que lo que hace es introducir el comando ***cls*** en la consola de Windows, limpiando así la pantalla de la consola. Justo después definimos los booleanos ***running*** (que representa si el programa esta ejecutando o no) y ***new_user_choice*** que representa si el usuario quiere elegir un nuevo usuario. Después definimos un bloque ***while*** que engloba a todo el resto del código de esta función, con el objetivo de cuando cambiemos el booleano ***running*** al valor ***False*** finalice de esta manera el programa.

Lo primero que se hará es elegir el usuario, para ello, comprobaremos el booleano ***new_user_choice*** mencionado anteriormente, si este tiene el valor ***True*** procederemos a elegir usuario mediante la función ***user_choice()*** y acto seguido conectarnos a la base de datos mediante ***connection_establishment()***. En el caso de que ocurra un error al conectarnos a la base de datos, se avisará de ello, se pondrá el booleano ***running*** a ***False*** y así acabaremos la ejecución del programa.

En caso de que no haya ningún error y, por tanto, estemos conectados a la base de datos, pediremos al usuario una consulta mediante la variable ***sql_query***, que consta de un input en forma de string, cuando el usuario haya introducido su consulta, presionará enter y se pasa a analizar el tipo de consulta que ha insertado.

Para analizar la consulta se separa mediante un ***split()*** que lo que hace es dado el string de parámetro, definirá cuando separar palabras y realizar una lista de manera que, cada palabra sea un elemento de la lista. Una vez que hemos conseguido la lista, vamos a la sentencia ***if*** donde miraremos la primera palabra de la consulta, que accedemos a ella mediante la orden ***sql_query_splitted[0]***, y si esta es igual que *select* o *SELECT* (ya que el lenguaje de programación Python distingue entre mayúsculas y minúsculas) si esta condición se cumple, verificaremos que la consulta insertada es de tipo select, por tanto ejecutarmeos la consulta introducida mediante la función ***select_query()*** (que en esta función se hace *fetch* a la consulta, cosa que no se podría hacer si hacemos una consulta de tipo insert). Y si no se cumple la condición anteriormente mencionada, podemos decir que es una consulta de tipo insert o modificadora, por tanto la ejecutaremos mediante la función ***insert_query()*** (que hace *commit* para guardar los cambios, cosa que en una consulta observadora o de tipo select no tendría sentido, ya que no se modifica nada)

Una vez ejecutada la consulta, se le preguntará al usuario si quiere hacer más consultas, y este responderá con un *si* o un *no*, por tanto se hará uso de la función ***yes_or_no_choice()***, en el caso de que el usuario introduzca un no, terminaremos al conexión mediante la función ***connection_termination()***, pondremos el booleano ***running*** al valor ***False*** y aparecerá un aviso por pantalla diciendo que el programa ha terminado ejecución. En el caso de que se quieran hacer más consultas, seremos preguntados también si queremos cambiar de usuario, y haremos uso, de nuevo, de la función ***yes_or_no_choice()***, ya que el usuario tendrá que responder con si o no, en el caso de que responda que sí, nos iremos al bloque ***else*** y pondremos el booleano ***new_user_choice*** al valor ***True***, y en el caso de que responda que no, pondremos el booleano ***new_user_choice*** al valor ***False*** y haremos un ***rollback*** a la conexión, para evitar errores de transacciones entre consultas SQL.

Si en algún momento, cancelamos la ejecución del programa mediante teclado usando (CTRL + C) también aparecerá un mensaje por pantalla avisando de que el programa ha sido finalizado mediante teclado. El código de la función main, se muestra a continuación:

```
def main():

    try:
        os.system('cls')
        running = True
        new_user_choice = True

        while (running):

            if (new_user_choice):
                try:
                    connection_establishment(user_choice())
                except (psycopg2.OperationalError):
                    print('Error en la conexion a la base de datos')
                    running = False
                    print('\n\n\t\t-----------------=[PROGRAMA FINALIZADO
POR FALLO DE CONEXION]=-----------------\n\n')
                    break
            sql_query = str(input('\nIntroduce la consulta a realizar:
'))
            sql_query_splitted = sql_query.split(' ')
            if (sql_query_splitted[0].__eq__('select' or 'SELECT')):
                select_query(sql_query)
            else:
                insert_query(sql_query)
            print('\n\n¿Desea hacer mas consultas? (si / no): ')
            if yes_or_no_choice().__eq__(False):
                print('\n\nCerrando...')
                connection_termination()
                time.sleep(0.5)
                print('\n\t\t-----------------=[PROGRAMA FINALIZADO]=----
-------------\n\n')
                running = False
            else:
                print('\n\n¿Desea cambiar de usuario? (si / no)')
                if yes_or_no_choice().__eq__(False):
                    new_user_choice = False
                    connection.rollback()
                else:
                    new_user_choice = True
            os.system('cls')

    except KeyboardInterrupt:
        print('\n\n\t\t-----------------=[PROGRAMA FINALIZADO POR
TECLADO]=-----------------\n\n')
```

Por último, se llama a la función ***main()*** para que el programa funcione.

```
main()
```