

# Desarrollo de Interfaces



KK

I.E.S. Carrillo Salcedo

Trabajo realizado por Equipo 01:

2º Desarrollo de Aplicaciones

Molero Marín, Juan

Multiplataforma

Rodríguez López, Julio

Curso 2024/2025

Martínez Lorda, Julián

Tema 02 – Boletín 03

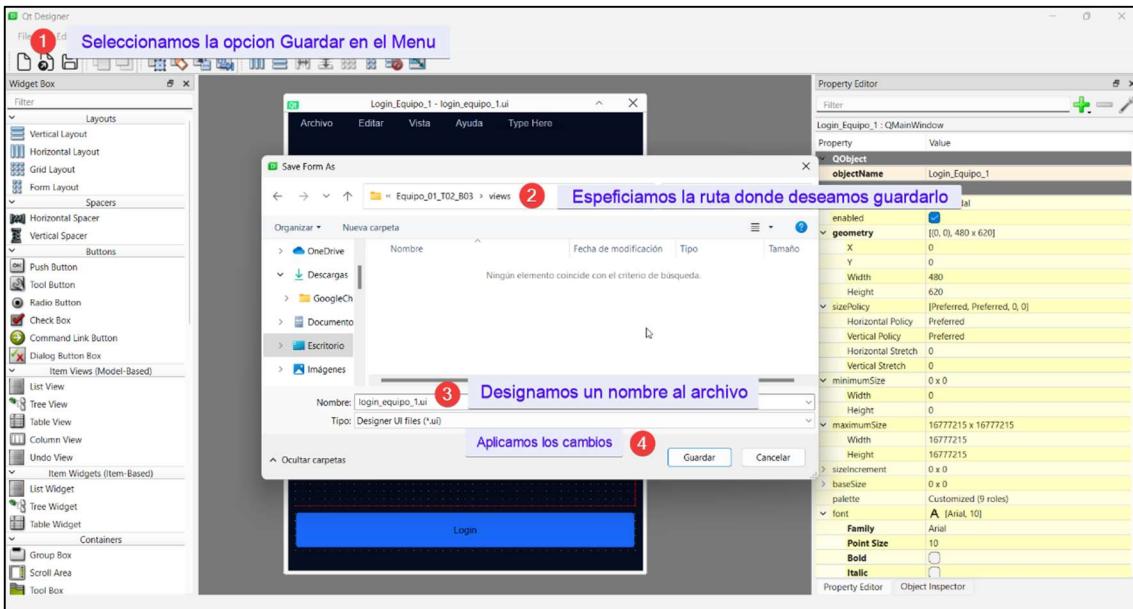
## Contenido

1.- Integración de la interfaz creada con QT Designer (archivo .ui) en Python:.....	3
2.- Integración de Eventos, Señales y Slots a la Interfaz en Python: .....	6
2.1.- Slots implementados en el Registro: .....	6
2.2.- Slots implementados en el Login:.....	7
3.- Integración de la Base de Datos dentro de la Aplicación: .....	9
4.- Resumen sobre la Aplicación:.....	11
5.- Despliegue de la Aplicación:.....	13
5.1.- Entorno Virtual:.....	13
5.2.- Convertir Archivo .ui en .py:.....	15
5.3.- Docker y BBDD PostgreSQL: .....	17
6.- Bibliografía: .....	21

### 1.- Integración de la interfaz creada con QT Designer (archivo .ui) en Python:

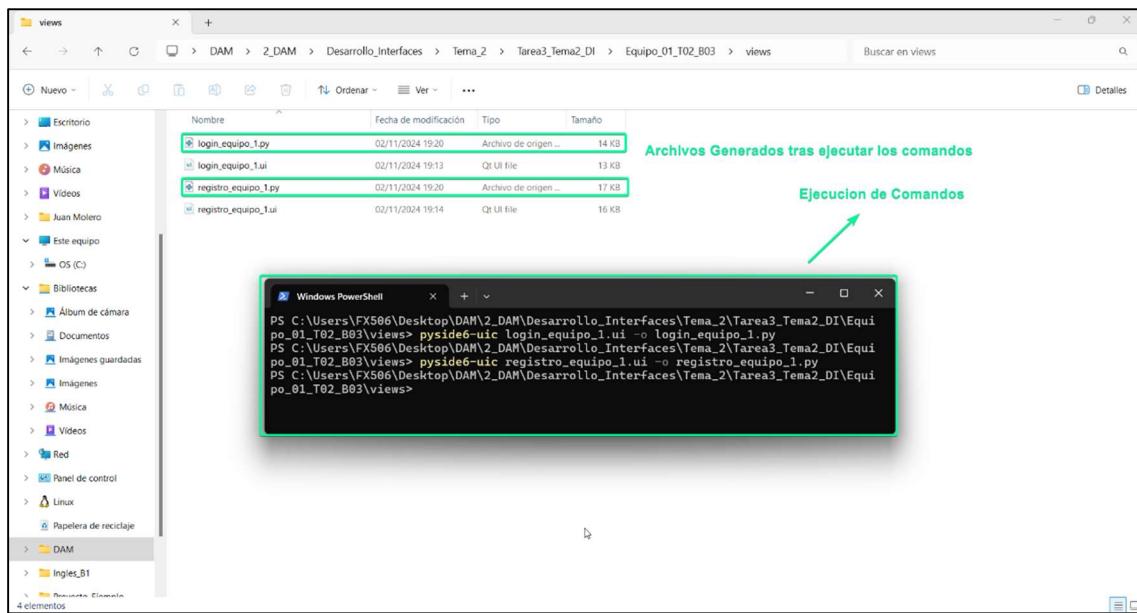
Nuestro primer paso ha sido acceder a la plataforma QT Designer, desde la cual obtendremos nuestros dos archivos .ui respectivamente: el de la ventana de Login y el de la ventana de Registro.

Para guardar el archivo seguiremos los siguientes pasos:



Una vez realicemos esto con nuestros dos archivos ('login\_equipo\_1.ui' y 'registro\_equipo\_1.ui'), mediante la consola Powershell accederemos a la ruta donde se encuentran, y posteriormente **se ejecutará el siguiente comando** para generar dos archivos .py dentro del mismo directorio:

**pyside6-uic 'archivo.ui' -o 'archivo.py'** → Donde archivo.ui será el archivo que deseamos convertir y archivo.py el nuevo archivo generado, con el nombre que nosotros le especifiquemos. De esta forma ejecutamos los comandos de la siguiente forma:



En nuestro caso concreto, los hemos generado dentro del directorio 'views' en el proyecto actual, por lo cual si los abrimos con VSC, y se crea un archivo Python independiente que los ejecute (main.py), podremos visualizar nuestras interfaces con éxito respectivamente. Cabe destacar que se le ha dado una mejor legibilidad y se les han incluido comentarios a ambos archivos, con el objetivo de poder trabajar posteriormente de forma más cómoda con ellos. Por tanto, aquí podemos ver el resultado:

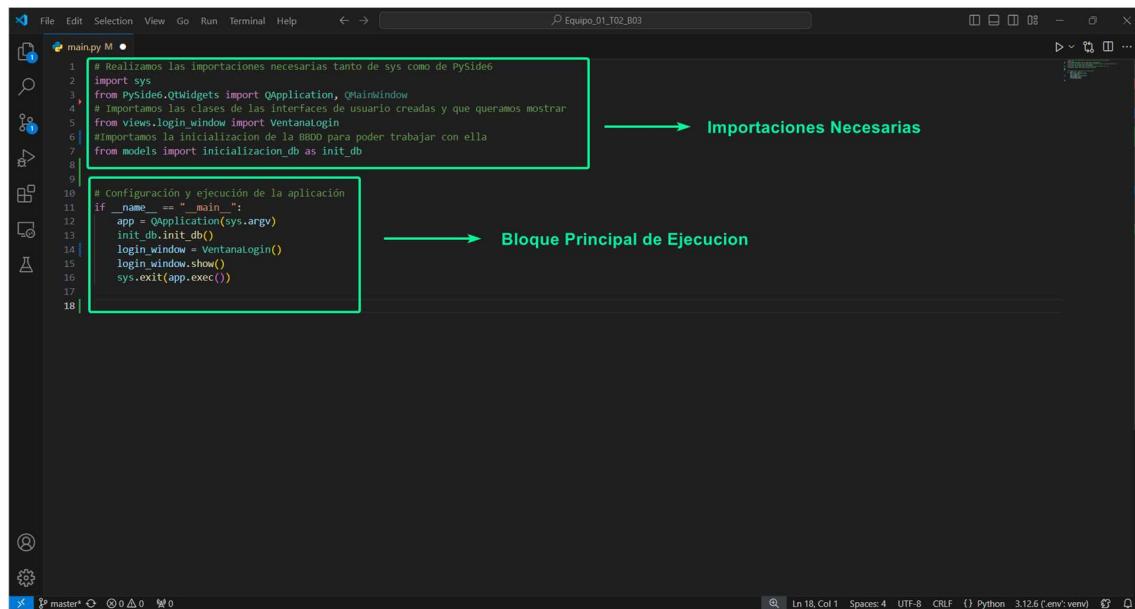
The screenshot shows the VS Code interface with the project 'Equipo\_01\_T02\_B03' open. The 'EXPLORER' view shows the file structure: 'registro.equipo\_1.py' and 'login.equipo\_1.py'. The 'CODE' view shows the code for 'login.equipo\_1.py'. A green arrow points from the bottom left towards the code editor. The code is annotated with green boxes containing comments: '#Importaciones de módulos necesarios de PySide6', '#Clase para la interfaz de la ventana de login', '#Color de fondo y fuente establecidos dentro de nuestra ventana de login', and '#Definición de las acciones dentro de la barra de menú (Salir, Guardar, ... )'. A green box also highlights the text 'Añadimos Comentarios al Código para una mejor legibilidad'.

```

1 Importaciones de módulos necesarios de PySide6
2 #QTCORE, QTGUI y QWIDGETS incluyen componentes esenciales de QT como layouts, colores, ...
3 from PySide6.QtCore import (QCoreApplication, QDateTime, QLocale,
4     QObject, QPoint, QRect,
5     QSize, QTime, QVariant, QT)
6 from PySide6.QtGui import (QAction, QBrush, QColor, QConicalGradient,
7     QIcon, QImage, QKeySequence, QLinearGradient,
8     QPainter, QPalette, QPixmap, QRadialGradient,
9     QTransform)
10 from PySide6.QtWidgets import (QApplication, QFormLayout, QGridLayout, QHBoxLayout,
11     QLabel, QVBoxLayout, QLineEdit, QMainWindow,
12     QMenuBar, QPushButton, QSizePolicy,
13     QStatusBar, QTabWidget, QWidget)
14
15 #Clase para la interfaz de la ventana de login
16 class UI_Login_Equipo_1(QObject):
17     #Metodo que nos configurara la interfaz de usuario para la ventana de login
18     def setupUI(self, login_Equipo_1):
19         #Configuración básica de nuestra ventana principal de login: se configura el nombre del objeto si no ha sido establecido
20         if not login_Equipo_1.objectName():
21             login_Equipo_1.setObjectName("Login_Equipo_1")
22         login_Equipo_1.resize(480, 620)
23         login_Equipo_1.setLayoutDirection(Qt.LayoutDirection.LeftToRight)
24         login_Equipo_1.setAutoFillBackground(False)
25
26         #Color de fondo y fuente establecidos dentro de nuestra ventana de login
27         login_Equipo_1.setStyleSheet("background-color: #f0f0f0; font: 10pt \"Arial\";")
28         login_Equipo_1.setTabShape(QTabWidget.TabShape.Rounded)
29
30         #Definición de las acciones dentro de la barra de menú (Salir, Guardar, ...)
31         self.actionNuevo = QAction(login_Equipo_1)
32         self.actionNuevo.setObjectName("actionNuevo")
33         self.actionNuevo.setObjectName("actionNuevo")
34         self.actionNuevo.setObjectName("actionNuevo")
35         self.actionNuevo.setObjectName("actionNuevo")
36         self.actionNuevo.setObjectName("actionNuevo")
37         self.actionNuevo.setObjectName("actionNuevo")
38         self.actionNuevo.setObjectName("actionNuevo")

```

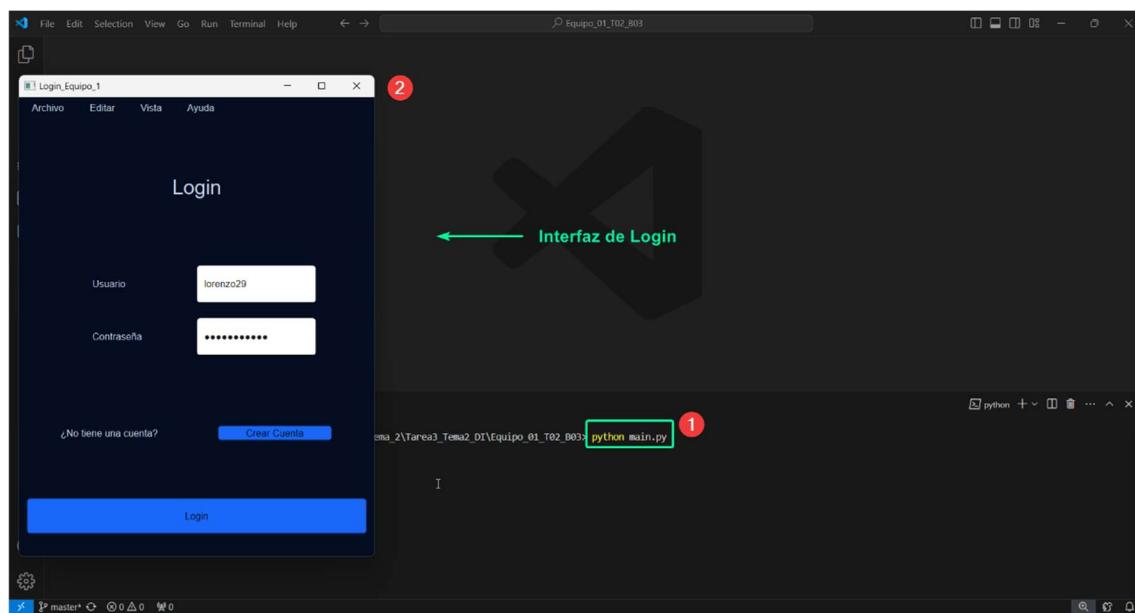
El archivo que nos permitirá ejecutar cada una de las dos interfaces generadas, por otra parte, es éste:



```
# Realizamos las importaciones necesarias tanto de sys como de PySide6
import sys
from PySide6.QtWidgets import QApplication, QMainWindow
# Importamos las clases de las interfaces de usuario creadas y que queramos mostrar
from views.login.window import VentanaLogin
#Importamos la inicialización de la BBDD para poder trabajar con ella
from models import inicializacion_db as init_db

# Configuración y ejecución de la aplicación
if __name__ == "__main__":
    app = QApplication(sys.argv)
    init_db()
    login_window = VentanaLogin()
    login_window.show()
    sys.exit(app.exec_())
```

Finalmente, para comprobar que el proceso ha salido con éxito, debemos de irnos a la consola de nuestro VSC, y ejecutar el comando **python main.py** desde la raíz del proyecto, que es donde tenemos ubicado dicho archivo. Una vez ejecutado, se abrirá la ventana de Login de nuestra futura aplicación, y nuestro siguiente objetivo será el de conectarla a la ventana de Registro mediante slots y señales:



## 2.- Integración de Eventos, Señales y Slots a la Interfaz en Python:

En el caso de nuestro equipo, hemos conseguido la integración de eventos mediante la creación de Slots en nuestras clases `login_window` y `registro_window`, las cuáles manejan la lógica de las clases QT donde tenemos nuestras interfaces.

### 2.1.- Slots implementados en el Registro:

Dentro del Registro, hemos implementado dos Slots diferentes:

```
@slot()
def mostrar_ventana_login(self):
    """
    Método que se ejecuta al hacer clic en 'Iniciar sesión'.
    Oculta esta ventana de registro y muestra la ventana de login.
    """

    # Oculta la ventana de registro --> ventana actual donde nos situamos
    self.hide()

    # Si existe una ventana padre, la utiliza para abrir la ventana de login
    if self.parent() is not None: # Verifica que haya una ventana padre
        # Crea una nueva instancia de la ventana de login
        ventana_login = self.parent() # Suponiendo que la ventana de login es la ventana padre
        ventana_login.show() # Muestra la ventana de login
```

Con este primer Slot, ocultaremos la ventana actual donde nos encontramos, se creará posteriormente una instancia de la ventana de Registro, mediante parent (estamos heredando de QMainWindow), y se terminará mostrando.

```
@slot()
def procesar_registro_usuario(self):
    """
    Método para gestionar el registro de un usuario nuevo.
    Comprueba que las contraseñas coincidan y delega la creación al controlador.
    """

    # Obtiene los datos introducidos en los campos de texto
    correo = self.ui.email_valor.text()
    nombre_usuario = self.ui.usuario_valor.text()
    contrasena = self.ui.password_valor.text()
    contrasena_confirmacion = self.ui.password_confirmada_valor.text()

    # Verifica que las contraseñas coincidan
    if contrasena != contrasena_confirmacion: # Compara ambas contraseñas
        # Muestra un mensaje de error si las contraseñas no coinciden
        print("Las contraseñas no coinciden")
        return # Sale del método si las contraseñas no son iguales

    # Intenta registrar al nuevo usuario utilizando el controlador
    if self.controlador_usuario.register_user(correo, nombre_usuario, contrasena, contrasena_confirmacion):
        # El controlador verifica si el usuario ya existe y, de no ser así, lo guarda en la base de datos
        # Muestra un mensaje de éxito en la consola si el registro fue exitoso
        print("Usuario registrado con éxito")
    else:
        # Si ocurre algún problema (usuario ya existente o error en la base de datos), se muestra un mensaje aquí
        print("Error al registrar el usuario")
```

Con este segundo Slot, procesaremos el registro de un usuario cuando haya llenado todos los campos, y las contraseñas coincidan respectivamente. En caso afirmativo, dicho usuario quedará registrado dentro de la BBDD, y podrá tener después una

posterior interacción iniciando sesión en la otra ventana. Cabe destacar que este Slot interactúa con un método del controlador (register\_user), parte que explicaremos más adelante, el cual es un módulo encargado de conectarnos la vista con los modelos y permitir que todo el flujo de la aplicación tenga éxito.

### Conexión de los Slots:

Estos dos slots serán conectados en el constructor principal de la clase con los elementos ui respectivos de las vistas. Además, también es donde se creará la instancia del controlador:

The screenshot shows a code editor with the following code:

```
6
7
8 class VentanaRegistro(QMainWindow):
9     def __init__(self, parent = None):
10         """
11             Constructor de la clase VentanaRegistro.
12
13             :param ventana_padre: Referencia a la ventana de login que actúa como ventana principal.
14             """
15
16         super().__init__(parent)
17         # Inicializa la ventana QMainWindow
18         # Crea una instancia de la interfaz gráfica de la ventana de registro
19         self.ui = Ui_ventana_registro()
20         # Configura la interfaz gráfica llamando al método setupUi
21         self.ui.setupUi(self)
22
23         # Conecta el botón "Iniciar sesión" para que al pulsarlo vuelva a la ventana de login
24         self.ui.boton_login.clicked.connect(self.mostrar_ventana_login)
25
26         # Conecta el botón "Registrar Cuenta" al método que se encarga de la creación de un usuario nuevo
27         self.ui.boton_crear_cuenta.clicked.connect(self.procesar_registro_usuario)
28
29         # Crea un controlador para manejar las operaciones relacionadas con el usuario en la base de datos
30         # Este controlador realiza tareas de validación y creación en el modelo de usuario y la base de datos
31         self.controlador_usuario = UsuarioController()
32
33
34
```

The code is annotated with two boxes:

- A blue box labeled "Contractor de la Clase" highlights the constructor definition and its parameters.
- A blue box labeled "Conexión de los Slots" highlights the two signal-slot connections: one for the login button and one for the register button.

### 2.2.- Slots implementados en el Login:

Dentro del módulo login\_window, por otra parte, nos encontramos con otros dos slots respectivamente:

The screenshot shows the implementation of the `abrir_ventana_registro` slot:

```
@Slot()
def abrir_ventana_registro(self):
    """
        Método que se ejecuta al hacer clic en 'Crear Cuenta'.
    """
    try:
        #Ocultamos la ventana de login
        self.hide()
        #Si no existe la ventana de registro, la creamos mediante una instancia padre
        if self.ventana_registro is None:
            self.ventana_registro = VentanaRegistro(parent=self)
        #Mostramos la ventana de registro y controlamos cualquier error con la excepción
        self.ventana_registro.show()
    except Exception as e:
        print(f"Error al abrir la ventana de registro: {e}")
```

Tal y como pasaba anteriormente, este slots es similar, pero para redirigirnos hacia la otra ventana (Registro), por lo cual a ser asociados a dos elementos UI diferentes y configurarlos de tal forma que la ventana actual se esconda, acabamos obteniendo una conexión fluida entre ambas vista.

```
@slot()
def iniciar_sesion(self):
    """
    Método que se ejecuta al hacer clic en 'Login'.
    """

    try:
        # Obtener el email y la contraseña desde la interfaz
        username = self.ui.campo_usuario.text() # Campo donde se ingresa el email
        password = self.ui.campo_password.text() # Campo donde se ingresa la contraseña

        # Llamar al método verify_user para verificar las credenciales
        usuario = self.gestor_usuarios.verify_user_by_name(username, password)

        if usuario is not None:
            print(f"Bienvenido {usuario.nombre_usuario}") # Muestra el mensaje de bienvenida
        else:
            print("Credenciales no válidas") # Mensaje de error si las credenciales son incorrectas
    except Exception as e:
        print(f"Error al iniciar sesión: {e}") # Captura y muestra cualquier error que ocurra
```

Con este otro slot, el cual interactúa con otro método del módulo controlador, verificamos si los campos introducidos dentro de la ventana son válidos, es decir son de un usuario que ha sido previamente registrado. En caso afirmativo, el sistema enviará un mensaje de bienvenida junto a su ‘username’, y en caso contrario, se informará de que las credenciales no han sido válidas.

### Conexión de los slots:

Como en el caso anterior, los hemos tenido que asociar dentro del constructor de la clase a los elementos ui correspondientes de las vistas, tal y como se puede apreciar en la siguiente imagen:

```
class VentanaLogin(QMainWindow):
    """
    Clase que representa la ventana de inicio de sesión.
    """

    def __init__(self, parent=None):
        """
        Constructor que configura la ventana de inicio de sesión y sus componentes.
        """

        try:
            super().__init__(parent)
            self.ui = Ui_Login_Equipo_1()
            self.ui.setupUi(self)
            self.ui.button_crear_cuenta.clicked.connect(self.abrir_ventana_registro)
            self.ui.button_login_2.clicked.connect(self.iniciar_sesion)
            self.ventana_registro = None
            self.gestor_usuarios = UsuarioController()
        except Exception as e:
            print(f"Error al inicializar la ventana de login: {e}")
```

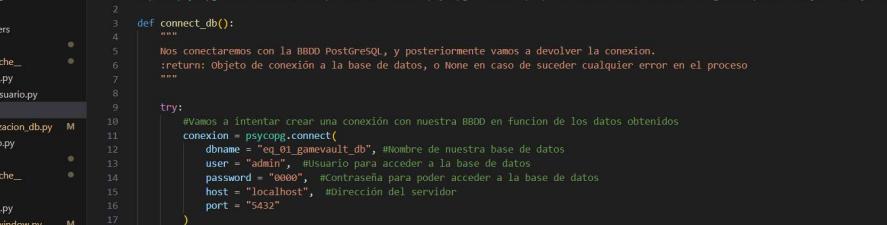
→ **Constructor de la Clase**

→ **Conexion de los Slots**

### **3.- Integración de la Base de Datos dentro de la Aplicación:**

La creación de todos los módulos para esta tarea se ha realizado dentro de la carpeta ‘models’, en la cual tenemos uno destinado a crear tanto la conexión de la base de datos como para cerrarla, así como otro enfocado en inicializar la misma, creando la tabla usuarios e insertando unos datos como ejemplo.

## Módulo db:



The screenshot shows a code editor with a Python file named `db.py` open. The code implements a connection manager for a PostgreSQL database. It includes functions for establishing a connection, handling exceptions, and closing the connection. The code uses the `psycopg` library and follows best practices like returning `None` for errors and printing error messages.

```
import psycopg # Necesitaremos importar el módulo 'psycopg' --> nos proporciona la biblioteca de PostgreSQL para trabajar con Python

def connect_db():
    """
    Nos conectaremos con la BBDD PostgreSQL, y posteriormente vamos a devolver la conexión.
    :return: Objeto de conexión a la base de datos, o None en caso de suceder cualquier error en el proceso
    """
    try:
        #Vamos a intentar crear una conexión con nuestra BBDD en función de los datos obtenidos
        conexion = psycopg.connect(
            dbname = "eq_01_gamewallet_db", #nombre de nuestra base de datos
            user = "admin", #usuario para acceder a la base de datos
            password = "0000", #contraseña para poder acceder a la base de datos
            host = "localhost", #dirección del servidor
            port = "5432"
        )
        return conexion
    except Exception as error:
        #Si durante el proceso de conexión se produce algún error, lo controlaremos mediante esta excepción, informando mediante un mensaje
        print(f"Error al conectar con la BBDD --> {error}")
        return None

def close_connection(conexion):
    """
    Método que nos cerrará la conexión con la BBDD PostgreSQL.
    :param conexión: la conexión a la BBDD que queremos cerrar
    """
    #Lo primero será comprobar si la conexión existe
    if conexión is not None:
        try:
            #Cerraremos la conexión, y en el caso de que ocurra cualquier tipo de error, será controlado mediante la excepción
            conexión.close()
        except Exception as error:
            print(f"Error al cerrar nuestra conexión con la BBDD --> {error}")

I
```

## Módulo inicializacion db:

Además de todo ello, se ha creado un módulo con el objeto u instancia que manipularemos mediante las vistas, en este caso Usuario, así como un módulo CRUD con diferentes métodos para poder operar con la Base de Datos:

### Módulo usuario:

```

#Clase Usuario
class Usuario:
    def __init__(self,email, nombre_usuario, password):
        #Definimos el constructor de la clase Usuario
        #param email: el correo electronico que tendra cada usuario
        #param nombre_usuario: el nombre que poseera el usuario
        #param password: contraseña que poseera nuestro usuario
        ...
        #Almacenamos los atributos privados de la clase
        self.__email = email
        self.__nombre_usuario = nombre_usuario
        self.__password = password

    #Getters de la clase
    @property
    def get_email(self):
        #Getter para obtener el email del usuario
        :return: Correo electrónico del usuario
        ...
        return self.__email #Devuelve el email almacenado

    @property
    def get_nombre_usuario(self):
        #Getter para obtener el nombre del usuario
        :return: Nombre del usuario
        ...
        return self.__nombre_usuario

    @property
    def get_password(self):
        ...
        #Getter para obtener la contraseña del usuario
        :return: Contraseña del usuario
        ...
        return self.__password

    #Setters de la clase
    @property
    def set_email(self, nuevo_email):
        ...

```

### Muestra del Módulo CRUD:

```

class CRUDUsuario:
    """
    Clase que implementa operaciones CRUD para la tabla 'usuarios' en la base de datos
    """

    def insertar_usuario (self,email,nombre_usuario,password):
        """
        Metodo para insertar un nuevo usuario dentro de la BBDD
        :param email: Correo electronico del nuevo usuario, el cual sera nuestra clave Primaria
        :param nombre_usuario: Nombre de usuario con el que insertaremos dentro de la BBDD
        :param nueva_password: Contraseña para el nuevo usuario que insertaremos dentro del BBDD
        ...
        # Si establece la consulta SQL para insertar el nuevo usuario dentro de la BBDD
        query = """
        INSERT INTO usuarios (email, nombre_usuario, password)
        VALUES (%s,%s,%s)
        """

        # Conectamos la BBDD utilizando la funcion db_conectar()
        conexion = db.connect_db()
        if conexion is None: #Verificaremos de que la conexión exista antes de poder continuar
            return False

        try:
            #Mediante un cursor, ejecutaremos la consulta con los parametros, y mediante el commit confirmaremos que se han realizado los cambios
            with conexion.cursor() as cursor:
                cursor.execute(query, (email,nombre_usuario,password))
                conexion.commit()
            return True
        #Inte cualquier tipo de error, sera controlado por una excepcion
        except Exception as error:
            print("Error al insertar el usuario --> [error]")
        finally:
            #Finalmente, cerraremos la conexión
            db.close_connection(conexion)

    def seleccionar_usuario (self,email):
        """
        Metodo para obtener un usuario de la BBDD a traves de su email (clave primaria)
        :param email: Correo Electronico del usuario que se desea obtener
        :return: Devolvemos una instancia de la clase Usuario si la encontramos, en caso contrario, retornamos None
        ...
        #Indicamos la consulta mediante la cual queremos obtener el usuario correspondiente

```

```

File Edit Selection View Go Run Terminal Help ⏎ → Equipo_01_T02_B03
EXPLORER ... crud_usuario.py
EQIPO_01_T02_B03 ...
controllers 157
env 158
models 159
__pycache__ 160
crud_usuario.py 161
db 162
inicializacion_db.py M 163
usuario.py M 164
views 165
__pycache__ 166
qt 167
__init__.py 168
login_window.py M 169
registro_window.py M 170
gunicorn 171
docker-compose.yml 172
main.py 173
requirements.txt 174
175
176
177
178
179
180
181
182
183
184
185

def seleccionar_usuario_por_nombre(self, nombre_usuario):
    """
    Método para obtener un usuario de la BBDD a través de su nombre de usuario.

    :param nombre_usuario: Nombre de usuario del usuario que se desea obtener.
    :return: Devolvemos una instancia de la clase Usuario si la encontramos, en caso contrario, retornamos None.
    """

    # Consulta SQL para buscar el usuario por nombre de usuario
    query = "SELECT email, nombre_usuario, password FROM usuarios WHERE nombre_usuario = %s"

    # Conectar a la base de datos
    conexion = db.connect_db()
    if conexion is None:
        return None

    try:
        # Ejecutar la consulta SQL
        with conexion.cursor() as cursor:
            cursor.execute(query, (nombre_usuario,))
            resultado = cursor.fetchone()  # Obtener el primer resultado
            if resultado is not None:
                return Usuario(*resultado)  # Crear y retornar una instancia de Usuario
            return None
    except Exception as error:
        print(f"Error al obtener el usuario por nombre de usuario --> {error}")
    finally:
        db.close_connection(conexion)

```

Seleccionar Usuario por Nombre

De esta forma, se podrá crear e inicializar la Base de Datos, y mediante el controlador podremos manipular todos los métodos creados en el CRUD para poder ser conectados posteriormente con las vistas.

#### 4.- Resumen sobre la Aplicación:

La aplicación funciona de la siguiente forma → Todo comienza en la clase main.py, donde crearemos una instancias de la ventana de Login y otra para poder inicializar la base de datos. Al arrancar el flujo, saltaremos a dicha ventana:

```

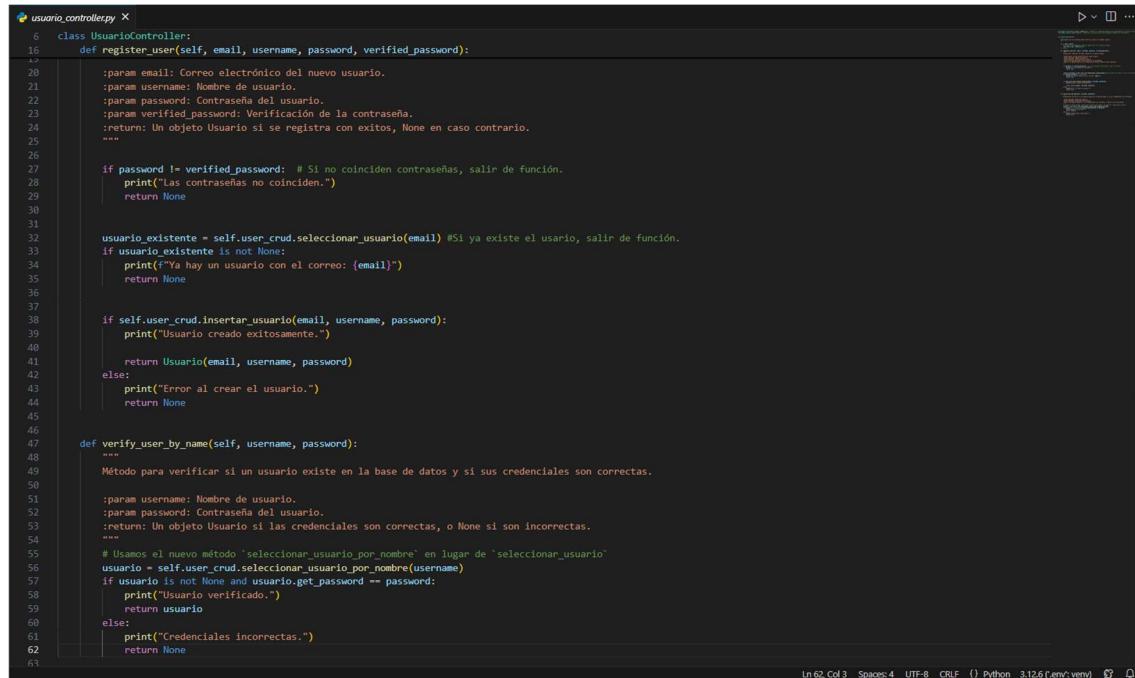
main.py
1 # Realizamos las importaciones necesarias tanto de sys como de PySide6
2 import sys
3 from PySide6.QtWidgets import QApplication, QMainWindow
4 # Importamos las clases de las interfaces de usuario creadas y que queremos mostrar
5 from views.login_window import VentanaLogin
6 from views.registro_window import VentanaRegistro
7 # Importamos la inicialización de la BBDD para poder trabajar con ella
8 from models import inicializacion_db as init_db
9
10
11 # Configuración y ejecución de la aplicación
12 if __name__ == "__main__":
13     app = QApplication(sys.argv)
14     init_db.init_db()
15     login_window = VentanaLogin()
16     login_window.show()
17     sys.exit(app.exec())
18
19

```

Importaciones Necesarias

Metodo de Ejecucion de la Clase

Una vez estemos en el Login, como se comentó anteriormente, nos podríamos redirigir a la ventana de Registro, o bien iniciar sesión. Tanto en un módulo como en otro, tenemos métodos que nos acaban llevando al módulo usuario\_controller:

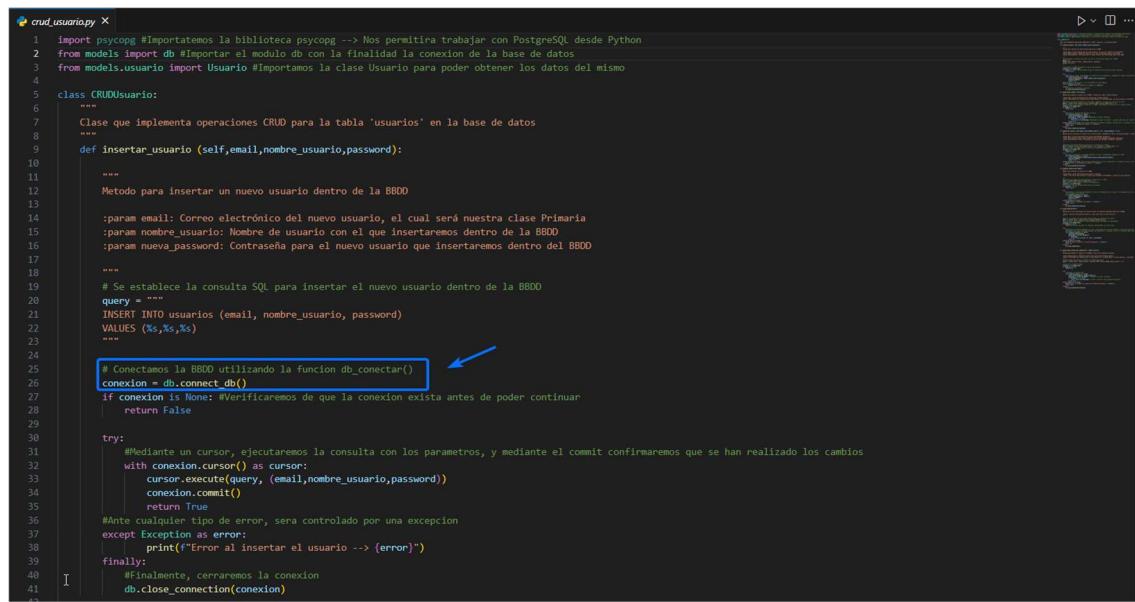


```

1  #!/usr/bin/python
2
3  class UsuarioController:
4      def register_user(self, email, username, password, verified_password):
5          """
6              :param email: Correo electrónico del nuevo usuario.
7              :param username: Nombre de usuario.
8              :param password: Contraseña del usuario.
9              :param verified_password: Verificación de la contraseña.
10             :return: Un objeto Usuario si se registra con éxito, None en caso contrario.
11         """
12
13         if password != verified_password: # Si no coinciden contraseñas, salir de función.
14             print("Las contraseñas no coinciden.")
15             return None
16
17         usuario_existente = self.user_crud.seleccionar_usuario(email) #Si ya existe el usuario, salir de función.
18         if usuario_existente is not None:
19             print("Ya hay un usuario con el correo: {email}")
20             return None
21
22
23         if self.user_crud.insertar_usuario(email, username, password):
24             print("Usuario creado exitosamente.")
25
26         return Usuario(email, username, password)
27     else:
28         print("Error al crear el usuario.")
29         return None
30
31
32     def verify_user_by_name(self, username, password):
33
34         """
35             Método para verificar si un usuario existe en la base de datos y si sus credenciales son correctas.
36
37             :param username: Nombre de usuario.
38             :param password: Contraseña del usuario.
39             :return: Un objeto Usuario si las credenciales son correctas, o None si son incorrectas.
40         """
41
42         # Usamos el nuevo método 'seleccionar_usuario_por_nombre' en lugar de 'seleccionar_usuario'
43         usuario = self.user_crud.seleccionar_usuario_por_nombre(username)
44         if usuario is not None and usuario.get_password == password:
45             print("Usuario verificado.")
46             return usuario
47         else:
48             print("Credenciales incorrectas.")
49             return None
50
51
52
53
54
55
56
57
58
59
5
60
61
62
63

```

Este será el módulo donde se gestione tanto el verificar un usuario (Login) como el registrarlo (Registro), el cual está utilizando métodos del módulo CRUD, específicamente el seleccionar usuario mediante el nombre y el insertar un usuario, respectivamente:



```

1  import psycopg2 #Importaremos la biblioteca psycopg --> Nos permitirá trabajar con PostgreSQL desde Python
2  from models import db #Importar el modulo db con la finalidad la conexión de la base de datos
3  from models.usuario import Usuario #Importamos la clase Usuario para poder obtener los datos del mismo
4
5  class CRUDUsuario:
6      """
7          Clase que implementa operaciones CRUD para la tabla 'usuarios' en la base de datos
8      """
9
10     def insertar_usuario (self,email,nombre_usuario,password):
11
12         """
13             Método para insertar un nuevo usuario dentro de la BBDD
14
15             :param email: Correo electrónico del nuevo usuario, el cual será nuestra clave Primaria
16             :param nombre_usuario: Nombre de usuario con el que insertaremos dentro de la BBDD
17             :param nueva_password: Contraseña para el nuevo usuario que insertaremos dentro del BBDD
18
19         """
20
21         # Se establece la consulta SQL para insertar el nuevo usuario dentro de la BBDD
22         query = """
23             INSERT INTO usuarios (email, nombre_usuario, password)
24             VALUES (%s,%s,%s)
25         """
26
27         # Conectamos la BBDD utilizando la función db.conectar()
28         conexion = db.connect_db()
29
30         if conexion is None: #Verificaremos de que la conexión exista antes de poder continuar
31             return False
32
33         try:
34             #Mediante un cursor, ejecutaremos la consulta con los parámetros, y mediante el commit confirmaremos que se han realizado los cambios
35             with conexion.cursor() as cursor:
36                 cursor.execute(query, (email,nombre_usuario,password))
37                 conexion.commit()
38                 return True
39
40             #Ante cualquier tipo de error, sera controlado por una excepción
41             except Exception as error:
42                 print("Error al insertar el usuario -> {error}")
43             finally:
44                 #Finalmente, cerraremos la conexión
45                 db.close_connection(conexion)
46
47
48
49
50
51
52
53
54
55
56
57
58
59
5
60
61
62
63

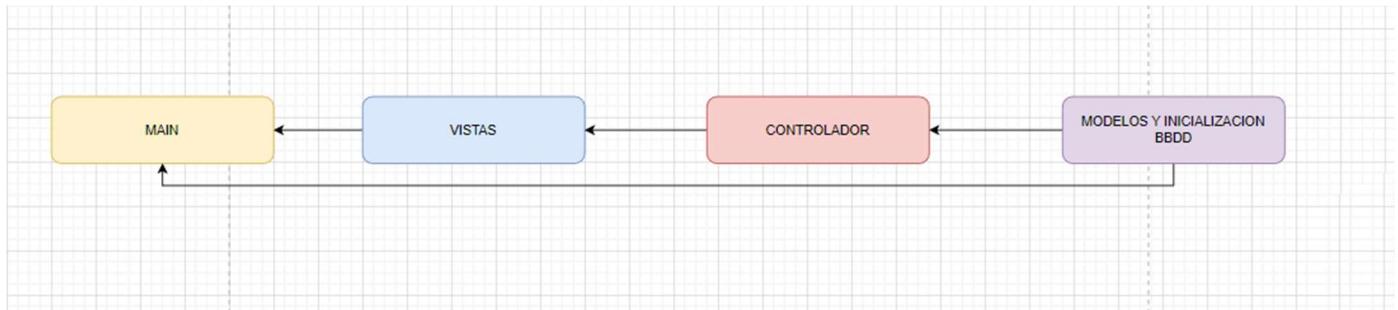
```

Aquí, apreciamos que se usan métodos de db, por lo que podemos afirmar que es el otro punto de origen en la aplicación. Por tanto, en conclusión, su funcionamiento podríamos decir que es el siguiente:

En los modelos, creamos 4 módulos → uno con el objeto/instancia a representar, uno para crear/iniciar la base de datos, otro para crear la conexión con los parámetros necesarios, y por último un módulo CRUD con el que se podrán realizar operaciones.

De forma posterior y mediante el Controlador, vamos a conectar métodos del CRUD con elementos de las vistas, al ser usados de forma final en los slots.

Es por eso que, desde nuestro main, instanciamos la primera ventana que queremos mostrar, creamos o inicializamos nuestra base de datos, y accedemos a la ventana que funciona según lo que acabamos de explicar.



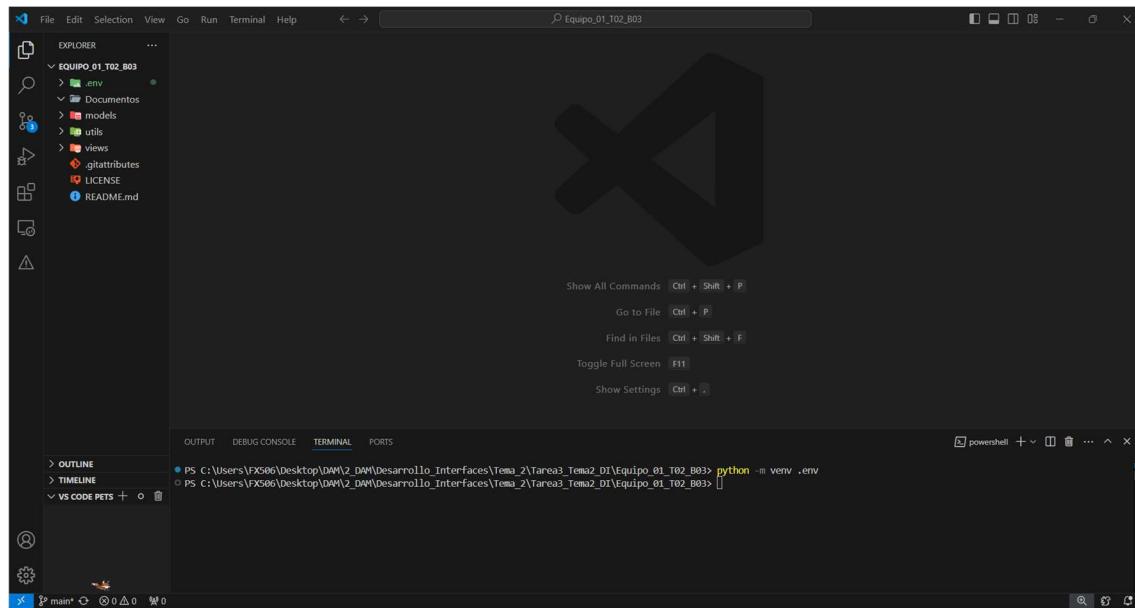
## 5.- Despliegue de la Aplicación:

### 5.1.- Entorno Virtual:

De forma previa a comenzar la realización de este proyecto, es necesario instalar el entorno virtual para poder trabajar, ya que entre otros aspectos fundamentales, trae integrado todas las dependencias que necesitaremos para poder desarrollar nuestro código, permitiendo instalar en cada proyecto lo necesario sin tener que interferir dentro de otros.

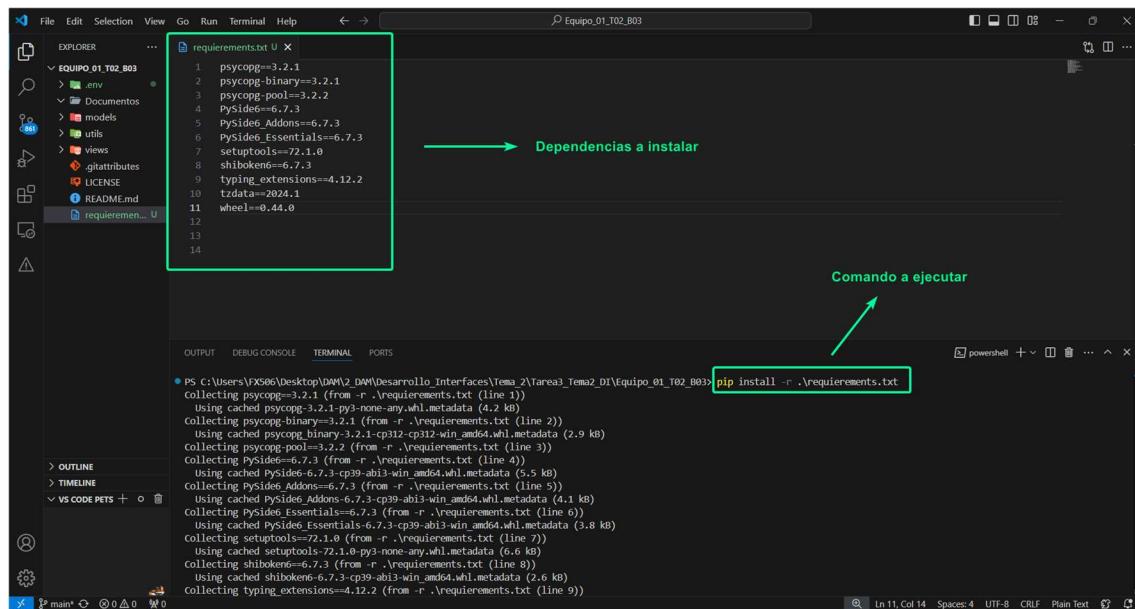
Para ello, lo primero que tenemos que hacer es crearnos un directorio, y desde VSC navegar hacia él. Una vez estemos dentro de la raíz del mismo, ejecutaremos el siguiente comando:

**python -m venv 'nombre del entorno':**



Una vez se haya creado el entorno virtual, lo siguiente será instalar las dependencias respectivamente, las cuáles se incluyen dentro de un archivo denominado “requirements.txt”, el cual deberá ser modificado en función de las especificaciones de cada proyecto, respectivamente. Es por ello que nos traemos dicho archivo a la raíz de nuestro proyecto, y posteriormente, ejecutamos el comando:

**pip install -r requirements.txt**

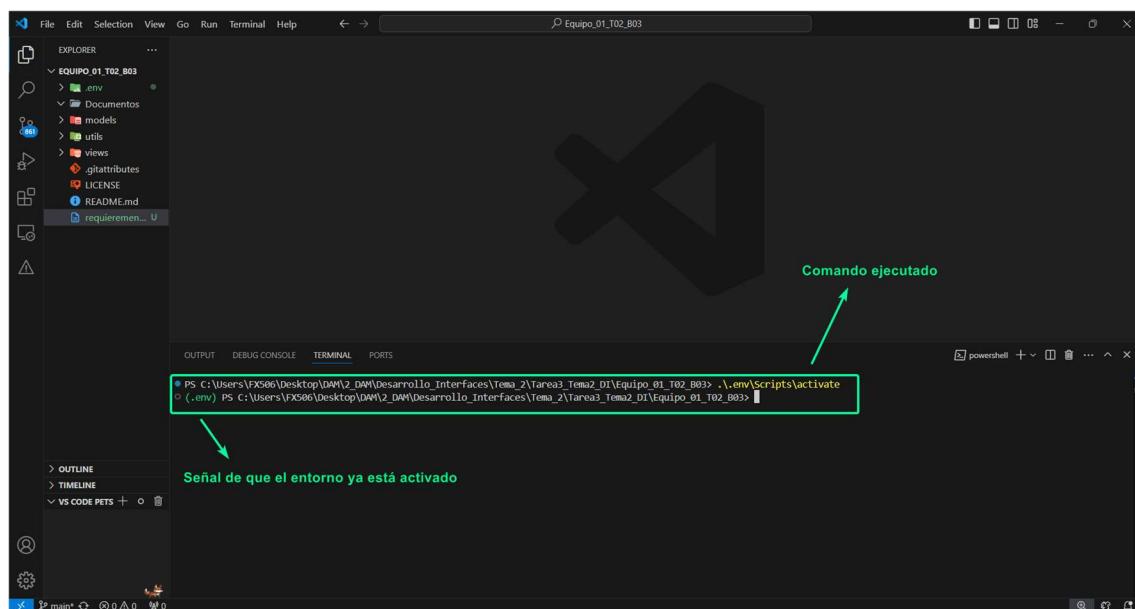


Una vez tengamos todas las dependencias que necesitamos instaladas, nuestro siguiente paso será el de activar el entorno previamente. Esto lo conseguimos mediante un comando, así como para desactivarlo y/o eliminarlo tendremos otros respectivamente:

`.\env\Scripts\activate` → Comando con el que activaremos el entorno virtual.

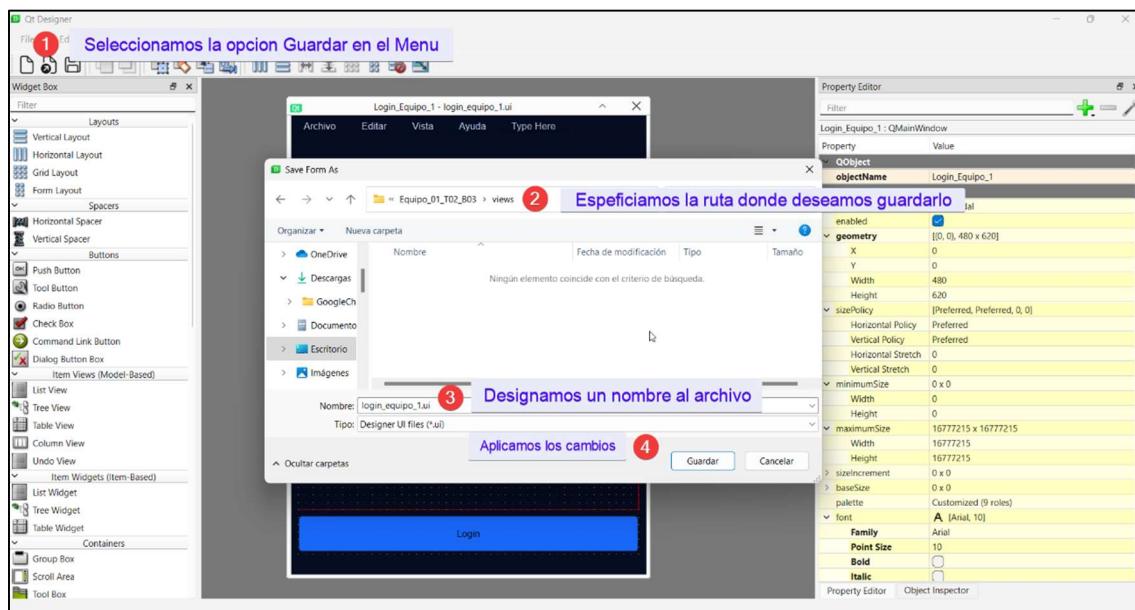
`deactivate` → Nos desactiva el entorno virtual.

`Remove-Item -Recurse -Force .\nombre_entorno` → Nos elimina el entorno virtual, para lo cual primero debe de ser desactivado.



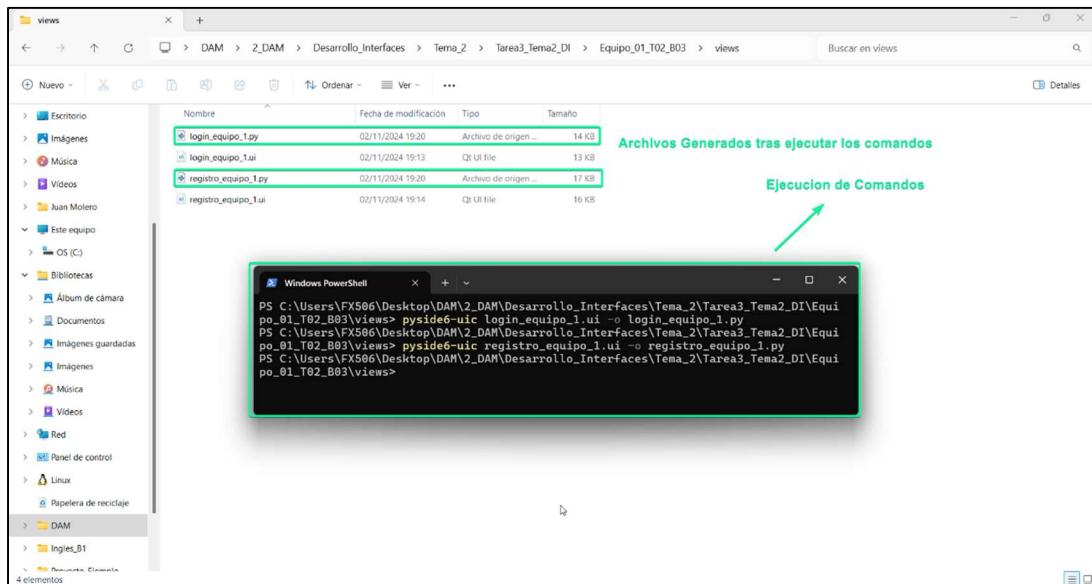
## 5.2.- Convertir Archivo .ui en .py:

Este paso ya fue explicado dentro del proyecto, sin embargo, vamos a volver a recordarlo brevemente: los archivos ui serán obtenidos desde Qt Designer de la siguiente forma:



Una vez guardados en el directorio donde queramos, el siguiente paso es ejecutar el siguiente comando para obtener un archivo .py a partir del .ui en el mismo directorio:

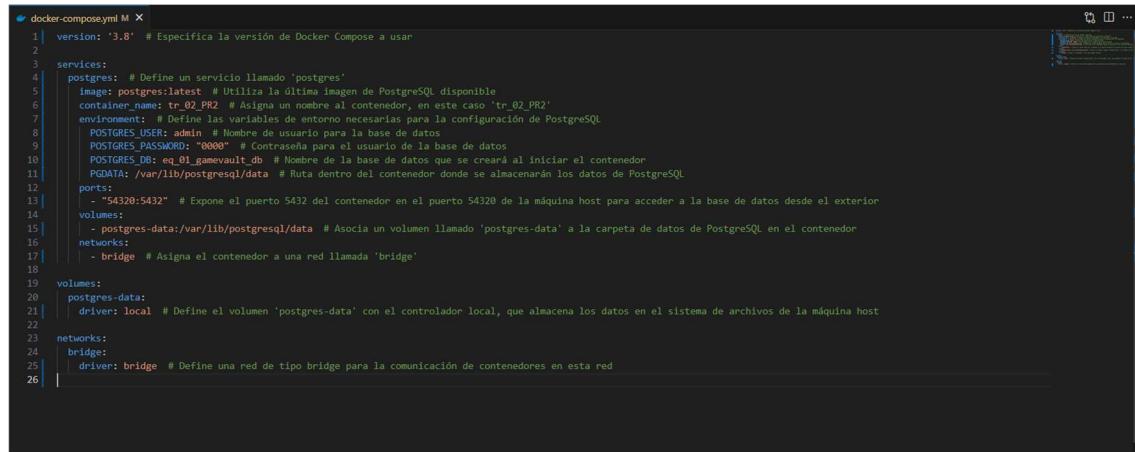
**pyside6-uic ‘archivo.ui’ -o ‘archivo.py’** → Donde archivo.ui será el archivo que deseamos convertir y archivo.py el nuevo archivo generado, con el nombre que nosotros le especifiquemos. De esta forma ejecutamos los comandos de la siguiente forma:



### 5.3.- Docker y BBDD PostgreSQL:

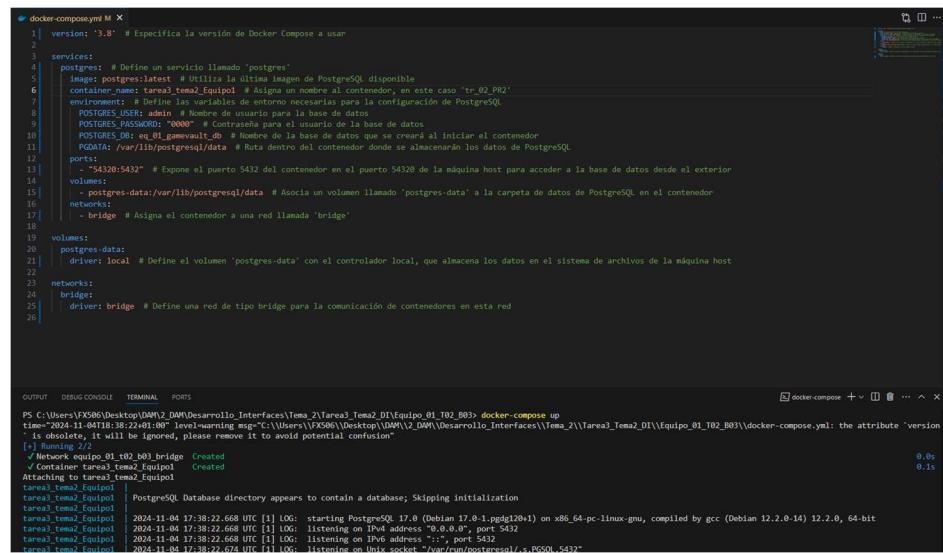
Para esta parte, se ha empleado, una herramienta de Docker denominada Docker Compose, la cual nos permite definir, ejecutar y gestionar respectivamente un conjunto de contenedores o un solo contenedor, con el objetivo de poder iniciarlos, detenerlos y/o configurarlos para que puedan gestionar el servicios postgres.

Es por ello que hemos tenido que realizar y configurar el archivo **docker-compose.yml** en la raíz del proyecto, mediante el cual con el comando **docker-compose up** podremos levantar y ejecutar nuestro contenedor de Docker, y mediante **docker-compose down** lo detendremos y eliminaremos posteriormente:



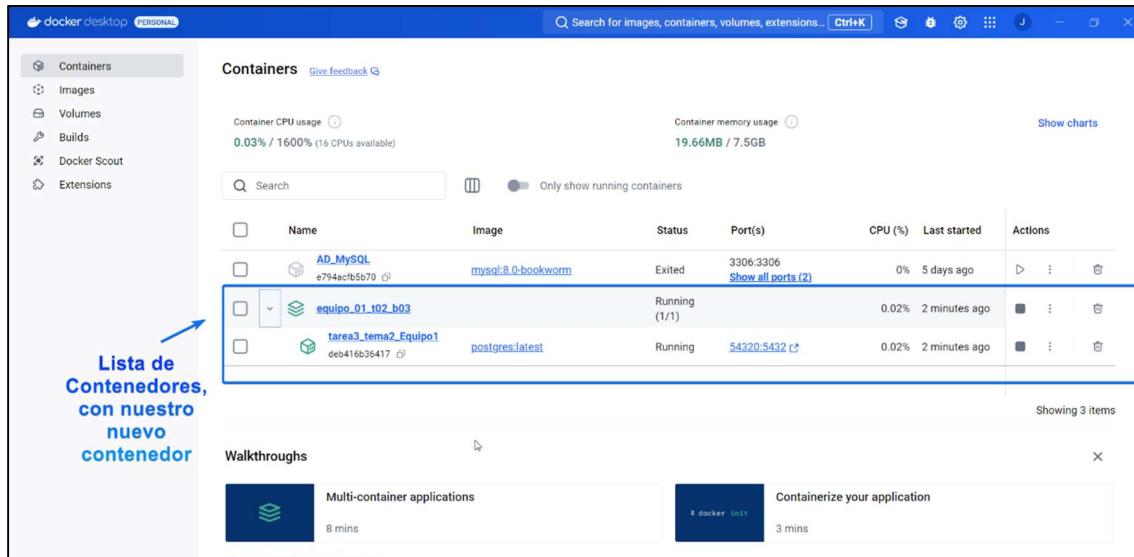
```
version: '3.8' # Especifica la versión de Docker Compose a usar
services:
  postgres: # Define un servicio llamado 'postgres'
    image: postgres:latest # Utiliza la última imagen de PostgreSQL disponible
    container_name: tr_02_PR2 # Asigna un nombre al contenedor, en este caso 'tr_02_PR2'
    environment:
      POSTGRES_USER: admin # Nombre de usuario para la base de datos
      POSTGRES_PASSWORD: "0000" # Contraseña para el usuario de la base de datos
      POSTGRES_DB: eq_01_gamewalut_db # Nombre de la base de datos que se creará al iniciar el contenedor
      PGDATA: /var/lib/postgresql/data # Ruta dentro del contenedor donde se almacenarán los datos de PostgreSQL
    ports:
      - "54320:5432" # Expone el puerto 5432 del contenedor en el puerto 54320 de la máquina host para acceder a la base de datos desde el exterior
    volumes:
      postgres-data:
        driver: local # Define el volumen 'postgres-data' con el controlador local, que almacena los datos en el sistema de archivos de la máquina host
    networks:
      - bridge # Asigna el contenedor a una red llamada 'bridge'
networks:
  bridge:
    driver: bridge # Define una red de tipo bridge para la comunicación de contenedores en esta red
```

Con el archivo ya configurado y listo, es hora de ejecutar el comando, crear e iniciar el contenedor, y por lo tanto, iniciar el servicio para poder tener conexión con la base de datos:



```
PS C:\Users\JXG06\Desktop\2_DAM\Desarrollo\Interfaces\Tema_2\Tarea3_Tema2_DI\Equipo_01_T02_B03> docker-compose up
time="2024-11-04T18:38:22+01:00" level=warning msg="C:\Users\JXG06\Desktop\2_DAM\Desarrollo_Interfaces\Tema_2\Tarea3_Tema2_DI\Equipo_01_T02_B03\docker-compose.yml: the attribute 'version' is obsolete, it will be ignored, please remove it to avoid potential confusion"
[1] 0s
✓ Network equipo_01_t02_b03_bridge Created
✓ Container tarea3_tema2_Equip01 Created
Attaching to tarea3_tema2_Equip01
tarea3_tema2_Equip01 | PostgreSQL Database directory appears to contain a database; Skipping initialization
tarea3_tema2_Equip01 | 2024-11-04 17:38:22.668 UTC [1] LOG: starting PostgreSQL 17.0 (Debian 17.0-1.postgres2024) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
tarea3_tema2_Equip01 | 2024-11-04 17:38:22.668 UTC [1] LOG: listening on IPv4 address "0.0.0.0", port 5432
tarea3_tema2_Equip01 | 2024-11-04 17:38:22.668 UTC [1] LOG: listening on IPv6 address "::", port 5432
tarea3_tema2_Equip01 | 2024-11-04 17:38:22.674 UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
```

## Resultado obtenido:



Con todo arrancado ya, es hora de probar dentro de nuestra aplicación, si nos deja insertar a un usuario y loguearlo posteriormente, así como consultar la lista de usuarios posteriormente en un software de gestión de Base de Datos, en este caso, **DBeaver**:

## Creación de Usuario:

```

version: '3.8' # Especifica la versión de Docker Compose
services:
  postgres: # Define un servicio llamado 'postgres'
    image: postgres:latest # Utiliza la última imagen de
    container_name: tr_02_PR2 # Asigna un nombre al conte
    environment: # Define las variables de entorno necesaria
      POSTGRES_USER: admin # Nombre de usuario para la base de datos
      POSTGRES_PASSWORD: "0000" # Contraseña para el usuario
      POSTGRES_DB: eq_01_gamewallet_db # Nombre de la base de datos
      PGDATA: /var/lib/postgresql/data # Ruta dentro del contenedor
    ports:
      - "54320:5432" # Expone el puerto 5432 del contenedor
    volumes:
      - postgres-data:/var/lib/postgresql/data # Asocia un volumen local con el contenedor
    networks:
      - bridge # Asigna el contenedor a una red llamada 'bridge'
  volumes:
    postgres-data:
      driver: local # Define el volumen 'postgres-data' con su tipo
  networks:
    bridge:
      driver: bridge # Define una red de tipo bridge para los servicios

```

Registro\_Equipo\_1

Registro

Email: juan@gmail.com

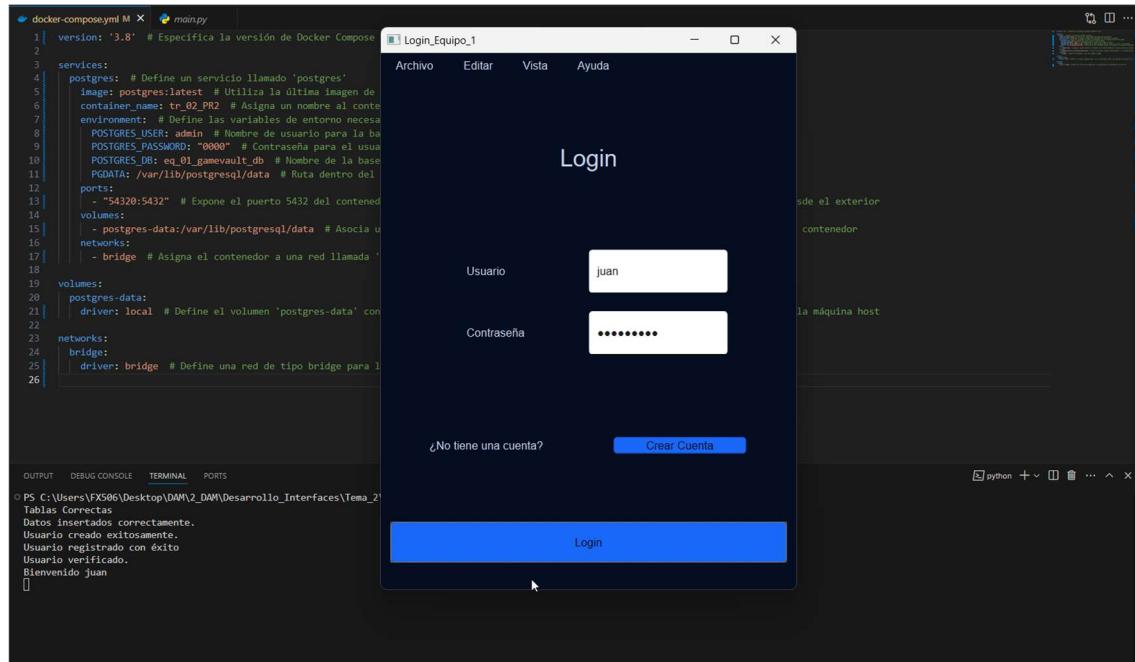
Usuario: juan

Contraseña: \*\*\*\*\*

Confirmar Contraseña: \*\*\*\*\*

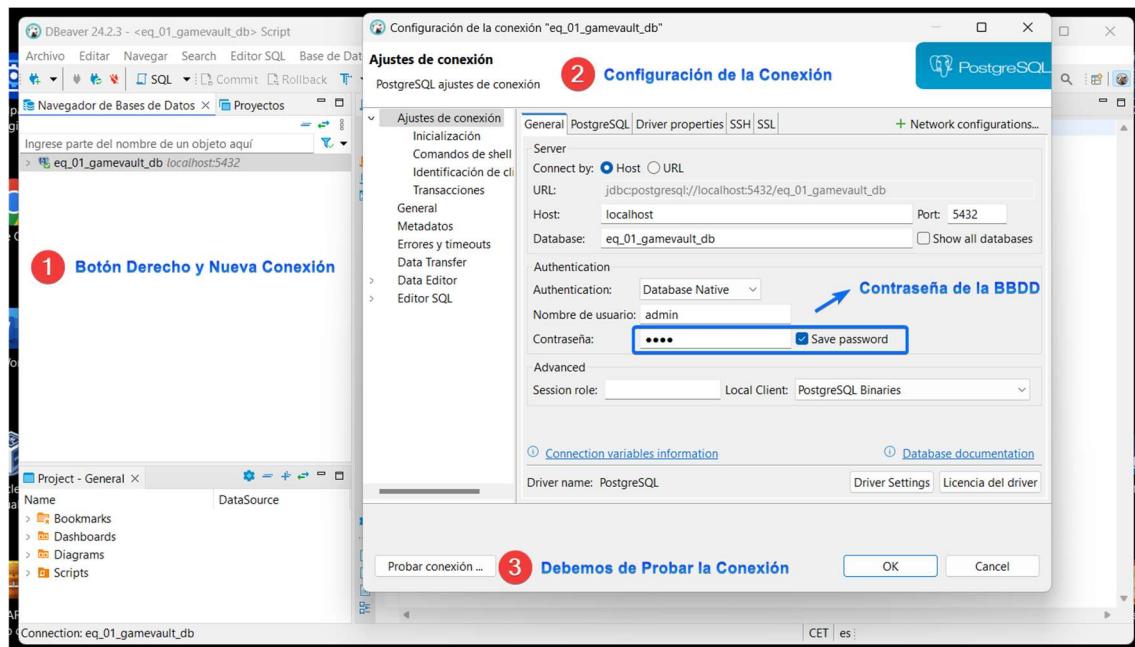
¿Ya tiene una cuenta?

### Login con el Usuario:

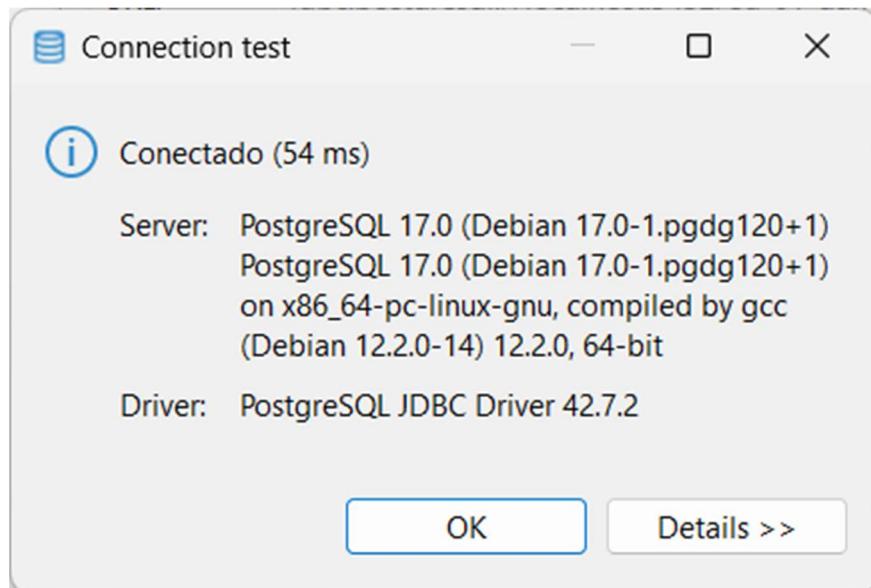


### Comprobación en DBeaver:

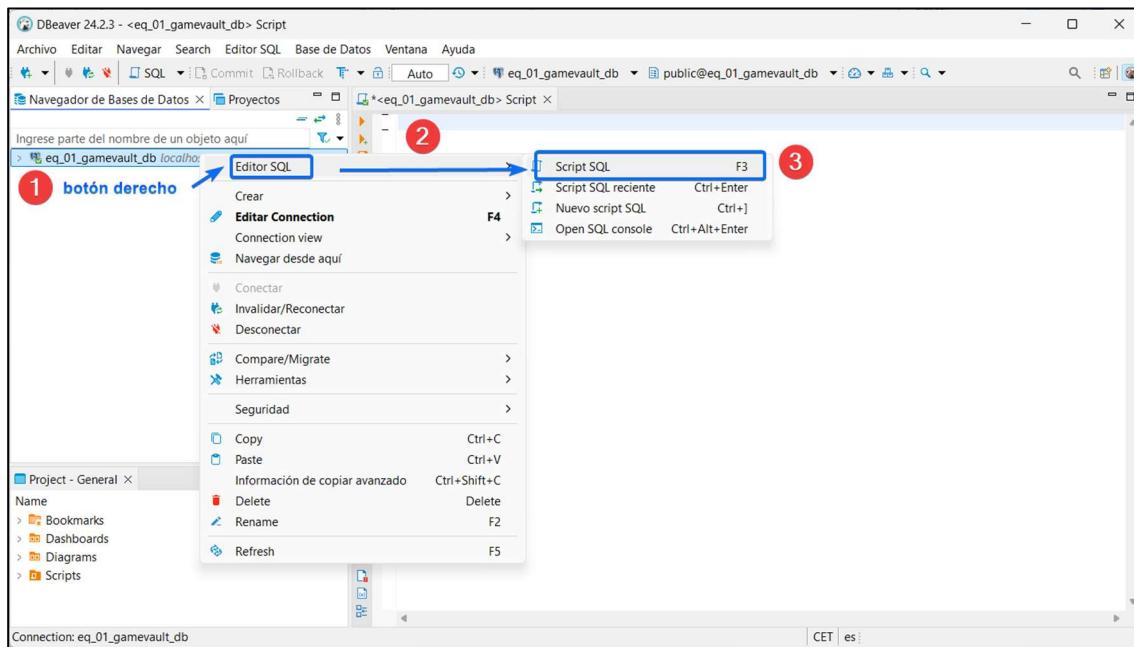
Para ello, tras acceder, tenemos que realizar lo siguiente:



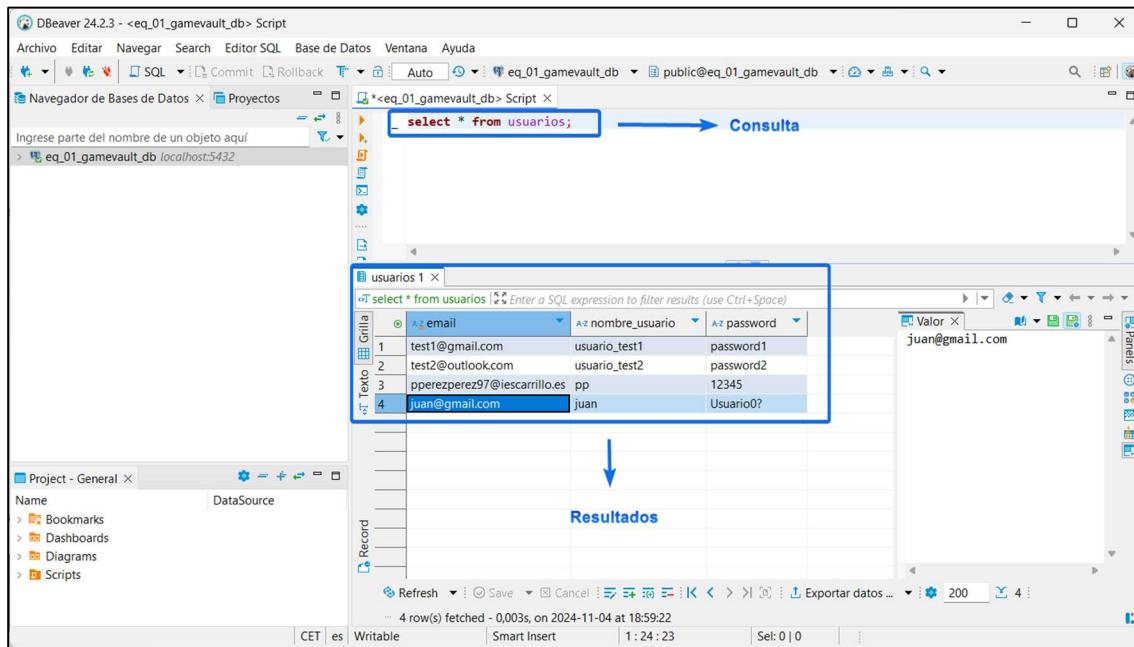
Si todo ha resultado de forma exitosa, deberías ver un mensaje de confirmación al darle a probar la conexión, tal y como éste:



Tras haber creado una conexión de forma exitosa, el siguiente paso sería acceder a la siguiente ruta para poder realizar cualquier consulta SQL:



Para terminar el proyecto y comprobar que todo ha sido un éxito, hemos realizado una consulta sencilla para listar a todos los usuarios: `SELECT * FROM usuarios`:



## 6.- Bibliografía:

Atoche Ortega, J. (2024). *Tema 02: Generación de interfaces gráficos con editores visuales y XML*. Dpto de Informática, IES Carrillo Salcedo.

*Qt Documentation*. (s. f.). Doc.qt.io. Recuperado 4 de noviembre de 2024, de <https://doc.qt.io/qt.html#qtforpython>