

Desarrollo interfaces

Tema 02 – Boletín 03



IES Carrillo Curso 24/25

Desarrollo de aplicaciones multiplataforma

Equipo 04:

Pedro González Martín

Antonio Gómez Camarena

David Castro Soriano

INDICE

Contenido

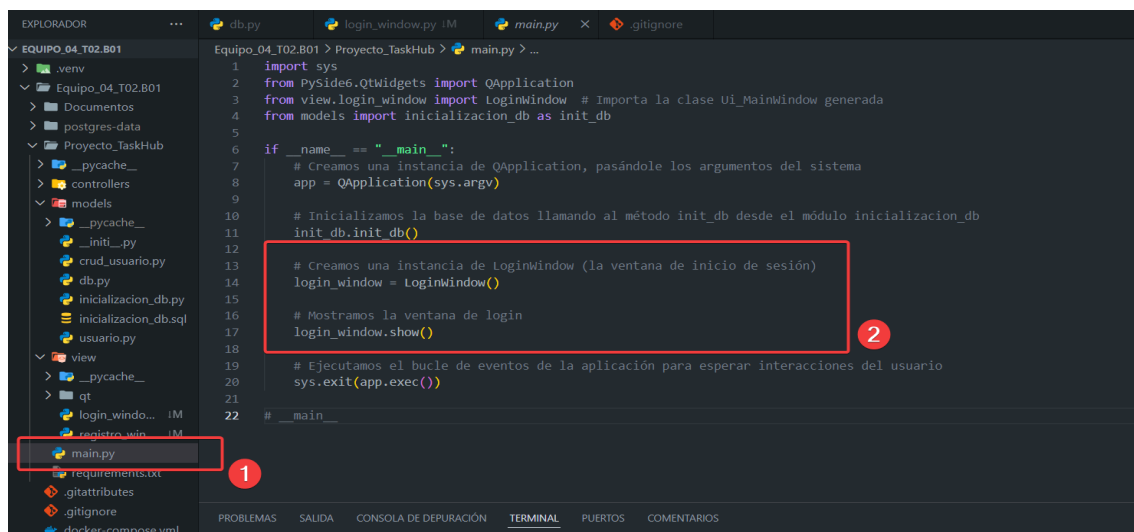
| | |
|---|----|
| Contenido..... | 1 |
| Integración de la interfaz creada con Qt Designer (archivo .ui) en Python..... | 3 |
| Integración de Eventos, Señales y Slots a la Interfaz en Python. | 4 |
| Integración de la BBDD en la aplicación..... | 6 |
| Resumen sobre la Aplicación | 13 |
| 1. Modelo (Models)..... | 13 |
| 2. Vista (Views) | 13 |
| 3. Controlador (Controllers | 13 |
| Integración en main.py | 14 |
| Flujo de la aplicación: | 14 |
| Inicio de la Aplicación (main.py): | 14 |
| Ventana de Inicio de Sesión (login_window.py)..... | 14 |
| Validación de Credenciales (en el Controlador)..... | 16 |
| Ventana de Registro (registro_window.py) | 16 |
| Proceso de Registro (en el Controlador) | 17 |
| Finalización | 18 |
| Despliegue de la aplicación..... | 18 |
| Como crear el entorno virtual en el proyecto | 18 |
| Como hacerlo por comandos | 22 |
| Como activarlo: | 22 |
| Que es el archivo requirements.txt y qué contiene el vuestro en concreto (explicar un poco que dependencias hay dentro) | 22 |
| Como instalar las dependencias desde el archivo requirements.txt..... | 23 |
| CONVERTIR ARCHIVO .UI EN .PY | 23 |
| DOCKER Y BBDD POSTGRESQL: | 24 |

| | |
|---|----|
| Mostrar como levantar el servicio de docker y que debe estar activo antes de continuar (esto en Windows es tan sencillo como abrir docker desktop)..... | 24 |
| Explicar que es docker-compose y para que se utiliza. | 26 |
| Explicar el contenido de dicho archivo para que se entienda. | 26 |
| Explicar cómo conectarnos a la bbdd con algún software de gestión de bbdd (podéis usar DBeaver u otro) | 27 |
| En PYTHON: | 29 |

Integración de la interfaz creada con Qt Designer (archivo .ui) en Python.

Creamos los .py correspondientes a cada ventana introduciendo el comando (pyside6-uic "clave.ui" -o "clave.py") donde clave.ui es la interfaz generada por Qt designer y clave.py es la clase Python generada a partir de esta interfaz.

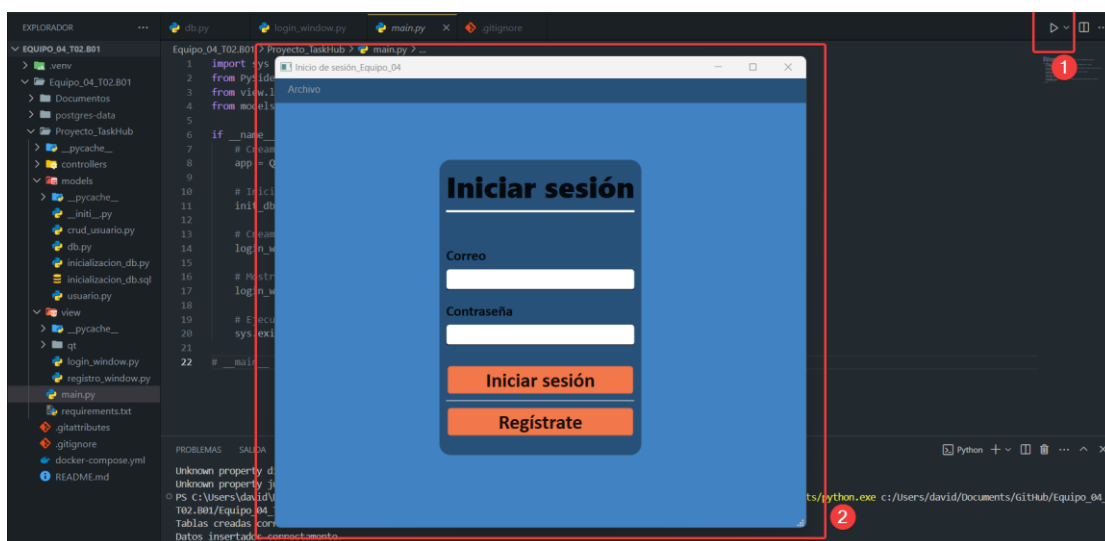
Lo segundo que debemos hacer es crear en Python una clase "main.py" en la que vamos a iniciar nuestra .py de login.



```
1 import sys
2 from PySide6.QtWidgets import QApplication
3 from view.login_window import LoginWindow # Importa la clase Ui_MainWindow generada
4 from models import inicializacion_db as init_db
5
6 if __name__ == "__main__":
7     # Creamos una instancia de QApplication, pasándole los argumentos del sistema
8     app = QApplication(sys.argv)
9
10    # Inicializamos la base de datos llamando al método init_db desde el módulo inicializacion_db
11    init_db.init_db()
12
13    # Creamos una instancia de loginWindow (la ventana de inicio de sesión)
14    login_window = LoginWindow()
15
16    # Mostramos la ventana de login
17    login_window.show()
18
19    # Ejecutamos el bucle de eventos de la aplicación para esperar interacciones del usuario
20    sys.exit(app.exec())
21
22 # __main__
```

1. Clase main.
2. Hacemos visible la interfaz con "login.show()".

Ya solo con estos pasos nos mostraría, al ejecutar el main.py la pantalla de login.



1. Ejecutamos pulsando el botón play o introduciendo el comando "Python ruta/main.py".
2. Vemos como se muestra la pantalla de Login.

Integración de Eventos, Señales y Slots a la Interfaz en Python.

```

from PySide6.QtWidgets import QMainWindow, QMessageBox # Importar la clase QMainWindow para crear la ventana principal
from PySide6.QtCore import Slot # Importar Slot para la conexión de señales y slots
from view.qt.Registro import Ui_Registro_Equipo04 # Importar la clase generada a partir del archivo .ui
from controllers.usuario_controller import UsuarioController # Importar el controlador para manejar las operaciones de registro y validación
import webbrowser

class RegistroWindow(QMainWindow):
    def __init__(self, parent=None):
        super().__init__(parent) # Llamar al constructor de la clase base QMainWindow
        self.ui = Ui_Registro_Equipo04() # Crear una instancia de la interfaz generada
        self.ui.setupUi(self) # Configurar la interfaz de usuario con el método setupUi

        self.ui.btn_iniciar_sesion.clicked.connect(self.function_volver_iniciar_sesion) 1
        self.ui.btn_registro.clicked.connect(self.function_registro) 2
        self.ui.vaciar_campos_de_texto.triggered.connect(self.vaciarCamposDeTexto) 3
        self.ui.action_nuestra_empresa_2.triggered.connect(self.abrirAcercaDe) 4

        self.usuario_controller = UsuarioController(5)

    # __init__

    @Slot() 1
    def function_volver_iniciar_sesion(self):
        self.hide()

        if self.parent() is not None:
            self.parent().mostrar_login()

    @Slot() 2
    def function_registro(self):
        nombre_usuario = self.ui.edit_usuario.text()
        email = self.ui.edit_correo.text()
        password = self.ui.edit_contrasenna.text()
        password_confirmada = self.ui.edit_r_contrasenna.text()

        if password != password_confirmada:
            print("Las contraseñas no coinciden")
            return

        print("Valores que se pasan a registrar_usuario:", email, nombre_usuario, password, password_confirmada)

        if self.usuario_controller.registrar_usuario(email, nombre_usuario, password, password_confirmada): 5
            print("usuario registrado exitosamente")
            mensaje_bienvenida = QMessageBox(self)
            mensaje_bienvenida.setWindowTitle("Registro exitoso")
            mensaje_bienvenida.setText("Usuario creado correctamente")
            mensaje_bienvenida.setIcon(QMessageBox.Information)
            mensaje_bienvenida.exec()
            self.vaciarCamposDeTexto()
        else:
            print("Error al crear el usuario")
            mensaje_error = QMessageBox(self)
            mensaje_error.setWindowTitle("Error Registro")
            mensaje_error.setText("Error al crear el usuario")
            mensaje_error.setIcon(QMessageBox.Critical)
            mensaje_error.exec()

    """
    Funcion que sirve para abrir una pagina del navegador
    """
    @Slot() 4
    def abrirAcercaDe(self):
        url = "https://github.com/dev10castro/Equipo_04_T02.B01/blob/main/README.md"
        webbrowser.open(url)
        # abrirAcercaDe

    """
    Funcion que sirve para eliminar todo el texto que haya por pantalla en los campos de registros
    """
    @Slot() 3
    def vaciarCamposDeTexto(self):
        print("Borrar textos")
        self.ui.edit_contrasenna.setText("")
        self.ui.edit_correo.setText("")
        self.ui.edit_r_contrasenna.setText("")
        self.ui.edit_usuario.setText("")
        # vaciarCamposDeTexto

```

1. Función que oculta la ventana registro y vuelve a mostrar la ventana de login.
2. En esta función obtenemos los datos de los campos de la ventana, comprueba que las contraseñas coincidan y llamamos al usuario controller para registrar un nuevo usuario, si se registra de forma correcta un QMessageBox nos informa de ello.
3. Función que borra lo introducido en los editText.
4. En esta función tenemos una variable que nos lleva hasta nuestro readme de GitHub, para ello hemos tenido que importar la librería de webbrowser.
5. Usuario controller es el encargado de introducir los datos de usuario en la BBDD.

```

from view.registro_window import RegistroWindow # Importar la clase de la ventana de registro
from controllers.usuario_controller import UsuarioController # Importar el controlador del usuario
import webbrowser

class LoginWindow(QMainWindow):
    """
    Clase que representa la ventana de inicio de sesión.
    """
    def __init__(self):
        """
        Constructor que inicializa la ventana de login y sus componentes.
        """
        super().__init__()
        self.ui = Ui_Inicio_Sesion_Equipo04()
        self.ui.setupUi(self)

        # Conectar las señales (clicks de botones) con los slots (métodos) correspondientes
        self.ui.boton_iniciar_sesion.clicked.connect(self.on_button_login_clicked)
        self.ui.boton_registrate.clicked.connect(self.on_button_crear_cuenta_clicked)
        self.ui.action_nuestra_empresa.triggered.connect(self.abrirAcercaDe)
        self.ui.vaciar_campo_texto.triggered.connect(self.vaciarCamposDeTexto)

        self.Ui_Registro_Equipo04 = None
        self.usuario_controller = UsuarioController()
        # __init__

    @slot()
    def on_button_login_clicked(self):
        print("Hemos pulsado el botón de login")
        name = self.ui.texto_usuario_correo.text()
        password = self.ui.texto_contrasenna.text()

        # Comprobamos de que el usuario y la contraseña existen
        if self.usuario_controller.verificar_usuario(name, password) is not None:
            print(f"Bienvenido {name}")

            # Crear el cuadro de diálogo de bienvenida
            mensaje_bienvenida = QMessageBox(self)
            mensaje_bienvenida.setWindowTitle("Inicio de sesión exitoso")
            mensaje_bienvenida.setText(f"Bienvenido, {name}!")
            mensaje_bienvenida.setIcon(QMessageBox.Information)
            mensaje_bienvenida.exec()
        else:
            print("Credenciales incorrectas")
            mensaje_error = QMessageBox(self)
            mensaje_error.setWindowTitle("Error inicio de sesión")
            mensaje_error.setText("Credenciales incorrecta")
            mensaje_error.setIcon(QMessageBox.Critical)
            mensaje_error.exec()

    @slot()
    def on_button_crear_cuenta_clicked(self):
        print("Hemos pulsado el botón de registro")
        self.hide()

        if self.Ui_Registro_Equipo04 is None:
            # Creamos la ventana de registro:
            self.Ui_Registro_Equipo04 = RegistroWindow(parent=self)

            # Mostramos la ventana de registro
            self.Ui_Registro_Equipo04.show()
        # on_button_crear_cuenta_clicked

    """
    Método para mostrar la ventana de login cuando se cierra la de registro.
    """
    @slot()
    def mostrar_login(self):
        # Mostrar la ventana de login
        self.show()
        # mostrar_login

    """
    Funcion que sirve para abrir una pagina del navegador
    """
    @slot()
    def abrirAcercaDe(self):
        url = "https://github.com/dev10castro/Equipo_04_T02.B01/blob/main/README.md"
        webbrowser.open(url)
        # abrirAcercaDe

    """
    Funcion que sirve para eliminar todo el texto que haya por pantalla en los campos de registros
    """
    @slot()
    def vaciarCamposDeTexto(self):
        print("Borrar textos")
        self.ui.texto_usuario_correo.setText("")
        self.ui.texto_contrasenna.setText("")
        # vaciarCamposDeTexto

```

1. Función que comprueba el correo y contraseña en la base de datos y si es correcto nos muestra un QMessageBox, en caso contrario nos mostrara otro avisando de que las credenciales son incorrectas.
2. Oculta la ventana login indicamos cual es la clase registro y la mostramos.
3. Nos lleva, al pulsar el botón “acerca de” a nuestro readme de GitHub.
4. Vacía los campos de texto.

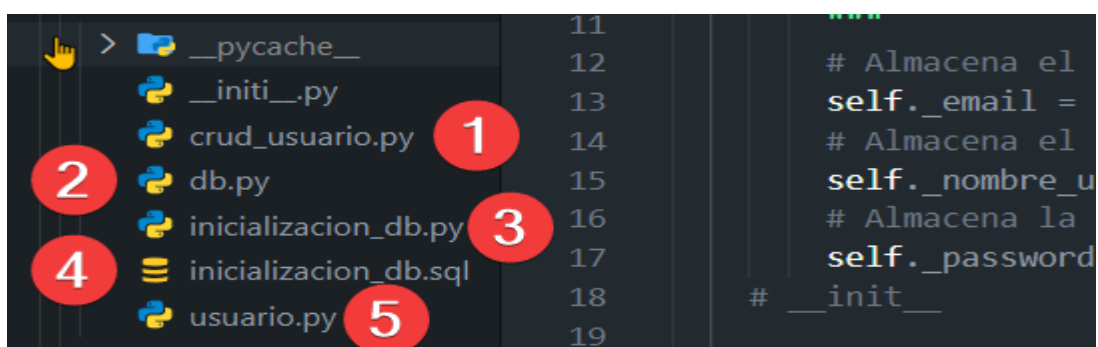
```

1  import sys
2  from PySide6.QtWidgets import QApplication
3  from view.login_window import LoginWindow # Importa la clase Ui_MainWindow generada
4  from models import inicializacion_db as init_db
5
6  if __name__ == "__main__":
7      # Creamos una instancia de QApplication, pasándole los argumentos del sistema
8      1 app = QApplication(sys.argv)
9
10     # Inicializamos la base de datos llamando al método init_db desde el módulo inicializacion_db
11     2 init_db.init_db()
12
13     # Creamos una instancia de LoginWindow (la ventana de inicio de sesión)
14     3 login_window = LoginWindow()
15
16     # Mostramos la ventana de login
17     4 login_window.show()
18
19     # Ejecutamos el bucle de eventos de la aplicación para esperar interacciones del usuario
20     5 sys.exit(app.exec())
21
22 # __main__

```

1. Crea una instancia de QApplication pasándole los argumentos del sistema.
2. Inicializa la base de datos.
3. Crea una instancia de loginwindows.
4. Muestra la ventana de login.
5. Espera interacciones con el usuario.

Integración de la BBDD en la aplicación.



1. Crud Usuario: Tenemos las posibles consultas que se necesitan para interactuar con la BBDD.

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

    def crear_usuario(self, email, nombre_usuario, password):
        """
        Método para crear un nuevo usuario en la base de datos.

        :param email: Correo electrónico del nuevo usuario (clave primaria).
        :param nombre_usuario: Nombre de usuario.
        :param password: Contraseña del usuario.
        :return: True si el usuario fue creado correctamente, False en caso de error.
        """
        # Definir la consulta SQL para insertar el nuevo usuario en la base de datos
        query = """
        INSERT INTO usuarios (email, nombre_usuario, password)
        VALUES (%s, %s, %s)
        """
        # Conectar a la base de datos utilizando la función connect_db
        conn = db.connect_db()
        if conn is None: # Si no hay conexión, no se puede continuar
            return False

        try:
            # Ejecutar la consulta SQL dentro de un bloque con el cursor
            with conn.cursor() as cursor:
                cursor.execute(query, (email, nombre_usuario, password)) # Ejecutar la consulta con los parámetros
                conn.commit() # Confirmar los cambios en la base de datos
                return True # Indicar que la operación fue exitosa
        except Exception as error:
            print(f"Error al crear usuario: {error}") # Mostrar el error en caso de excepción
            return False
        finally:
            db.close_connection(conn) # Cerrar la conexión al terminar

    # crear usuario

    def obtener_usuario(self, email):
        """
        Método para obtener un usuario de la base de datos por su email.

        :param email: Correo electrónico del usuario que se desea obtener.
        :return: Un objeto de la clase Usuario si se encuentra, None en caso contrario.
        """
        # Definir la consulta SQL para obtener un usuario por su email
        query = "SELECT email, nombre_usuario, password FROM usuarios WHERE email = %s"
        # Conectar a la base de datos utilizando la función connect_db
        conn = db.connect_db()
        if conn is None: # Si no hay conexión, no se puede continuar
            return None

        try:
            # Ejecutar la consulta SQL dentro de un bloque con el cursor
            with conn.cursor() as cursor:
                cursor.execute(query, (email,)) # Ejecutar la consulta con el parámetro email
                resultado = cursor.fetchone() # Obtener el primer resultado
                if resultado is not None: # Comparación explícita de que el resultado no es None
                    # Desempaquetar la tupla 'resultado' y pasa cada valor como un argumento separado
                    # (email, nombre_usuario, password) al constructor de la clase 'Usuario'
                    return Usuario(*resultado)
                return None # Si no hay resultados, retornar None
        except Exception as error:
            print(f"Error al obtener usuario: {error}") # Mostrar el error en caso de excepción
            return None
        finally:
            db.close_connection(conn) # Cerrar la conexión al terminar

    # obtener usuario

    def actualizar_usuario(self, email, nuevo_nombre_usuario=None, nueva_password=None):
        """
        Método para actualizar los datos de un usuario en la base de datos.

        :param email: Correo electrónico del usuario a actualizar (clave primaria).
        :param nuevo_nombre_usuario: Nuevo nombre de usuario (opcional).
        :param nueva_password: Nueva contraseña (opcional).
        :return: True si los datos se actualizaron correctamente, False en caso de error.
        """
        # Definir la consulta SQL para actualizar un usuario por su email
        query = "UPDATE usuarios SET nombre_usuario = %s, password = %s WHERE email = %s"
        # Conectar a la base de datos utilizando la función connect_db
        conn = db.connect_db()
        if conn is None: # Si no hay conexión, no se puede continuar
            return False

        try:
            # Ejecutar la consulta SQL dentro de un bloque con el cursor
            with conn.cursor() as cursor:
                cursor.execute(query, (nuevo_nombre_usuario, nueva_password, email)) # Ejecutar la consulta
                conn.commit() # Confirmar los cambios en la base de datos
                return True # Indicar que la operación fue exitosa
        except Exception as error:
            print(f"Error al actualizar usuario: {error}") # Mostrar el error en caso de excepción
            return False
        finally:
            db.close_connection(conn) # Cerrar la conexión al terminar

    # actualizar usuario
```

Aquí se hace la consulta para crear un usuario en la BBDD. En caso de no crearse, nos lanza una excepción.

Consulta para obtener un usuario de la BBDD. En caso de error nos lanza una excepción.

Consulta para hacer un update a un usuario.


```
def eliminar_usuario(self, email):
    """
    Método para eliminar un usuario de la base de datos.

    :param email: Correo electrónico del usuario a eliminar.
    :return: True si el usuario fue eliminado correctamente, False en caso de error.
    """
    # Definir la consulta SQL para eliminar un usuario por su email
    query = "DELETE FROM usuarios WHERE email = %s"
    # Conectar a la base de datos utilizando la función connect_db
    conn = db.connect_db()
    if conn is None: # Si no hay conexión, no se puede continuar
        return False

    try:
        # Ejecutar la consulta SQL dentro de un bloque con el cursor
        with conn.cursor() as cursor:
            cursor.execute(query, (email,)) # Ejecutar la consulta con el parámetro email
            conn.commit() # Confirmar los cambios en la base de datos
            return True # Indicar que la operación fue exitosa
    except Exception as error:
        print(f"Error al eliminar usuario: {error}") # Mostrar el error en caso de excepción
        return False
    finally:
        db.close_connection(conn) # Cerrar la conexión al terminar

# eliminar_usuario
```

Consulta pra eliminar un usuario de la BBDD.

```
def listar_usuarios(self):
    """
    Método para obtener una lista de todos los usuarios registrados en la base de datos.

    :return: Lista de objetos Usuario, o lista vacía en caso de error.
    """
    # Definir la consulta SQL para obtener todos los usuarios
    query = "SELECT email, nombre_usuario, password FROM usuarios"
    # Conectar a la base de datos utilizando la función connect_db
    conn = db.connect_db()
    if conn is None: # Si no hay conexión, no se puede continuar
        return []

    try:
        # Ejecutar la consulta SQL dentro de un bloque con el cursor
        with conn.cursor() as cursor:
            cursor.execute(query) # Ejecutar la consulta
            resultados = cursor.fetchall() # Obtener todos los resultados
            if resultados: # Verificar si `resultados` contiene datos
                # Retorna una lista de objetos Usuario
                return [Usuario(*row) for row in resultados]
            return [] # Retorna una lista vacía si no hay resultados
    except Exception as error:
        print(f"Error al listar usuarios: {error}") # Mostrar el error en caso de excepción
        return []
    finally:
        db.close_connection(conn) # Cerrar la conexión al terminar

# listar_usuarios
```

Consulta que nos muestra email, nombre, password de todos los usuarios.

CRUDUsuario

2. db.py: Clase que hace conexión con la base de datos, tiene los métodos conectar y cerrar conexión. Aquí están todos los parámetros necesarios para conectar con una base de datos postgresql.

```

Proyecto_TaskHub > models > db.py > connect_db
1 # Archivo: T02_PR01\models\db.py
2
3 import psycopg # Importamos el módulo `psycopg`, que es una biblioteca para trabajar con PostgreSQL desde Python.
4
5 # Función para conectar a la base de datos PostgreSQL
6 def connect_db():
7     """
8     Conectar a la base de datos PostgreSQL y devolver la conexión.
9
10    :return: Objeto de conexión a la base de datos o None en caso de error.
11    """
12    try:
13        # Intentamos crear una conexión a la base de datos usando los datos proporcionados
14        conn = psycopg.connect(
15            dbname="eq_04_taskHub_db", # Nombre de la base de datos 2
16            user="admin", # Usuario de la base de datos 3
17            password="0000", # Contraseña para acceder a la base de datos 4
18            host="localhost", # Dirección del servidor de base de datos (local) 5
19            port="54320" # Puerto en el que se ejecuta PostgreSQL 6
20        )
21        return conn # Si la conexión es exitosa, se devuelve el objeto `conn`
22    except Exception as error:
23        # Si ocurre algún error durante la conexión, se captura la excepción y se imprime un mensaje de error
24        print(f"Error al conectar con la base de datos: {error}")
25        return None # En caso de error, se devuelve `None` para indicar que no se pudo conectar
26
27 # connect_db
28
29 # Función para cerrar la conexión a la base de datos PostgreSQL
30 def close_connection(conn):
31     """
32     Cerrar la conexión a la base de datos PostgreSQL.
33
34     :param conn: La conexión a la base de datos que se desea cerrar.
35     """
36     if conn is not None: # Verificamos si la conexión existe (no es None)
37         try:
38             conn.close() # Intentamos cerrar la conexión si está activa
39         except Exception as error:
40             # Si ocurre algún error al cerrar la conexión, se captura la excepción y se imprime un mensaje de error
41             print(f"Error al cerrar la conexión a la base de datos: {error}")
42
43 # close_connection

```

3. inicializacionbd.py: Tiene los métodos create_tables(1), insert_data(2) e init_db(3), este último se encarga de llamar a los otros dos métodos para crear las tablas e insertar datos de ejemplo.

```

# Función para crear las tablas necesarias en la base de datos
def create_tables():
    """
    Crear las tablas de Usuarios si no existen en la base de datos.

    Se asegura de que la tabla 'usuarios' exista, de lo contrario la crea.
    """
    # Definir los comandos SQL que se ejecutarán para crear las tablas
    # Se usa 'CREATE TABLE IF NOT EXISTS' para evitar errores si la tabla ya existe

    # Campo 'email' como clave primaria
    # Campo 'nombre_usuario' obligatorio
    # Campo 'password' obligatorio
    commands = (
        """
        CREATE TABLE IF NOT EXISTS usuarios (
            email VARCHAR(255) PRIMARY KEY,
            nombre_usuario VARCHAR(255) NOT NULL,
            password VARCHAR(255) NOT NULL
        )
        """
    )

```

```

)

try:
    # Conectarse a la base de datos utilizando la función connect_db() del archivo db
    conn = db.connect_db()
    # Verificar si la conexión se realizó correctamente
    if conn is not None: # Hacemos explícita la condición
        with conn.cursor() as cur: # Usamos un cursor para ejecutar comandos SQL
            # Ejecutar cada comando dentro de la tupla 'commands'
            for command in commands:
                cur.execute(command) # Ejecuta el comando SQL
            conn.commit() # Confirmar (guardar) los cambios en la base de datos
            print("Tablas creadas correctamente.")
            conn.close() # Cerrar la conexión a la base de datos
        else:
            print("No se pudo conectar a la base de datos.")
    except Exception as error: # Capturar cualquier excepción que ocurra
        # Mostrar un mensaje de error si ocurre alguna excepción
        print(f"Error al crear las tablas: {error}")
# create_tables

```

```

# Función para insertar datos de ejemplo en la tabla 'usuarios'
def insert_data():
    """
    Insertar datos de ejemplo en la tabla 'usuarios' si no existen.

    Se insertan 3 usuarios de ejemplo y se utiliza 'ON CONFLICT' para evitar duplicados.
    """
    # Tupla que contiene los datos a insertar (email, nombre de usuario, contraseña)
    inserts = (
        ('usuario1@example.com', 'usuario1', 'usuario0?'),
        ('usuario2@example.com', 'usuario2', 'usuario0?'),
        ('usuario3@example.com', 'usuario3', 'usuario0?')
    )

    # Definir la consulta SQL para insertar los datos en la tabla 'usuarios'
    insert_query = """
        INSERT INTO usuarios (email, nombre_usuario, password)
        VALUES (%s, %s, %s)
        ON CONFLICT (email) DO NOTHING;
    """ # Se usa 'ON CONFLICT' para evitar errores de inserción si ya existe un usuario con el mismo email.

    try:
        # Conectarse a la base de datos
        conn = db.connect_db()
        # Verificar si la conexión se realizó correctamente
        if conn is not None: # Hacemos explícita la condición
            with conn.cursor() as cur: # Crear un cursor para ejecutar comandos SQL
                # Iterar sobre los usuarios en la tupla 'inserts'
                for user in inserts:
                    cur.execute(insert_query, user) # Ejecutar la consulta SQL para cada usuario
                conn.commit() # Confirmar (guardar) los cambios en la base de datos
                print("Datos insertados correctamente.")
            conn.close() # Cerrar la conexión a la base de datos
        else:
            print("No se pudo conectar a la base de datos.")
    except Exception as error: # Capturar cualquier excepción que ocurra
        # Mostrar un mensaje de error si ocurre alguna excepción
        print(f"Error al insertar datos: {error}")
# insert_data

```

```

# Función para inicializar la base de datos, creando tablas e insertando datos de ejemplo
def init_db():
    """
    Inicializar la base de datos llamando a las funciones de creación de tablas e inserción de datos.
    """
    create_tables() # Llamar a la función que crea las tablas
    insert_data() # Llamar a la función que inserta los datos de ejemplo
# init_db

```

4. inicialización_db.sql: Si la tabla usuarios no existe, la crea e inserta datos de prueba en la base de datos.

```
-- Crear la tabla de Usuarios si no existe
CREATE TABLE IF NOT EXISTS Usuarios (
    email VARCHAR(255) PRIMARY KEY, -- El email será la clave primaria
    nombre_usuario VARCHAR(100) NOT NULL,
    password VARCHAR(100) NOT NULL
);

-- Insertar datos de prueba en la tabla Usuarios
INSERT INTO Usuarios (email, nombre_usuario, password)
VALUES
('usuario1@example.com', 'usuario1', 'usuario0?'),
('usuario2@example.com', 'usuario2', 'usuario0?'),
('usuario3@example.com', 'usuario3', 'usuario0?');

-- Seleccionar todos los registros de la tabla Usuarios
SELECT * FROM Usuarios;

-- Eliminar todos los registros de la tabla Usuarios sin eliminar su estructura
TRUNCATE TABLE Usuarios;

-- Eliminar la tabla Usuarios si existe
DROP TABLE IF EXISTS Usuarios;
```

5. usuario.py: Clase objeto Usuario, con su constructor, sus getters y setter y los métodos `__str__` y `__repr__`.

```
class Usuario:
    def __init__(self, email, nombre_usuario, password):
        """
        Constructor para la clase Usuario.

        :param email: Correo electrónico del usuario.
        :param nombre_usuario: Nombre de usuario.
        :param password: Contraseña del usuario.
        """
        # Almacena el email del usuario (atributo privado)
        self._email = email
        # Almacena el nombre de usuario (atributo privado)
        self._nombre_usuario = nombre_usuario
        # Almacena la contraseña del usuario (atributo privado)
        self._password = password
    # __init__

    # Getters
    @property
    def get_email(self):
        """
        Getter para obtener el email del usuario.
        :return: Correo electrónico del usuario.
        """
        return self._email # Devuelve el email almacenado
    # get_email

    @property
    def get_nombre_usuario(self):
        """
        Getter para obtener el nombre de usuario.
        :return: Nombre de usuario.
        """
        return self._nombre_usuario # Devuelve el nombre de usuario almacenado
    # get_nombre_usuario
```

```

@property
def get_password(self):
    """
    Getter para obtener la contraseña del usuario.
    :return: Contraseña del usuario.
    """
    return self._password # Devuelve la contraseña almacenada
# get_password

# Setters
@get_email.setter
def set_email(self, nuevo_email):
    """
    Setter para establecer un nuevo email.
    :param nuevo_email: Nuevo correo electrónico.
    """
    self._email = nuevo_email # Establece un nuevo valor de email
# set_email

@get_nombre_usuario.setter
def set_nombre_usuario(self, nuevo_nombre_usuario):
    """
    Setter para establecer un nuevo nombre de usuario.
    :param nuevo_nombre_usuario: Nuevo nombre de usuario.
    """
    self._nombre_usuario = nuevo_nombre_usuario # Establece un nuevo valor de nombre de usuario
# set_nombre_usuario

@get_password.setter
def set_password(self, nueva_password):
    """
    Setter para establecer una nueva contraseña.
    :param nueva_password: Nueva contraseña.
    """
    self._password = nueva_password # Establece un nuevo valor de contraseña
# set_password

# Método para representar al usuario como una cadena
def __str__(self):
    """
    Representación en cadena del objeto Usuario.
    :return: Representación en formato de cadena del usuario.
    """
    return f"Usuario: {self.get_nombre_usuario}, Email: {self.get_email}" # Devuelve una representación legible
# __str__

# Método para representación oficial (útil para debugging y desarrollo)
def __repr__(self):
    """
    Representación formal del objeto Usuario, más detallada y útil para desarrolladores.
    :return: Representación en formato de cadena del usuario, incluyendo detalles internos.
    """
    return f"Usuario(email={self._email!r}, nombre_usuario={self._nombre_usuario!r}, password={self._password!r})"
# Devuelve una representación detallada para desarrolladores
# __repr__

# Método para cambiar la contraseña
def cambiar_password(self, nueva_password):
    """
    Método para cambiar la contraseña del usuario.
    :param nueva_password: Nueva contraseña del usuario.
    """
    self._password = nueva_password # Cambia la contraseña del usuario
# cambiar_password

```

**** Metodo __str__:**

Este método está diseñado para devolver una representación amigable para los usuarios finales. Se utiliza cuando se imprime el objeto o se convierte a cadena, por ejemplo, al usar ``print(objeto)``. En este caso, simplemente devuelve información básica del usuario como "Usuario: nombre, Email: email".

****Metodo `__repr__`:**

Este método devuelve una representación oficial del objeto, pensada para desarrolladores. Es útil para debugging o desarrollo, ya que muestra una representación más completa y precisa del estado interno del objeto. Se utiliza cuando se llama a ``repr(objeto)`` o en el intérprete interactivo.

Resumen sobre la Aplicación

Nuestro proyecto de aplicación TaskHub sigue el patrón MVC (Modelo-Vista-Controlador), lo que nos permite separar la lógica de datos, la interfaz gráfica y la conexión entre ambas, facilitando la organización y el mantenimiento del código.

1. Modelo (Models)

- El modelo es la capa que gestiona la información de los usuarios y la conexión con la base de datos. Aquí están los archivos que definen la estructura de los datos y las operaciones de base de datos, como el archivo `usuario.py`, que define la clase `Usuario`, y `crud_usuario.py`, que implementa las funciones para crear, leer, actualizar y eliminar usuarios en la base de datos. `db.py` configura la conexión a PostgreSQL, y `inicializacion_db.py` crea las tablas y carga datos de prueba para la aplicación.

2. Vista (Views)

- La vista es la interfaz de usuario y se encarga de mostrar las ventanas para iniciar sesión y registrarse. En `login_window.py`, se implementa la ventana de inicio de sesión, con botones y campos de texto para ingresar las credenciales. En `registro_window.py`, el usuario puede registrarse, ingresando sus datos, y la vista verifica que las contraseñas coincidan antes de pasar la información al controlador.

3. Controlador (Controllers)

- El controlador es el intermediario entre el modelo y la vista. En `usuario_controller.py`, gestiona las solicitudes de la vista, validando y coordinando la lógica antes de interactuar con el modelo. Por ejemplo, el controlador se asegura de que las contraseñas coincidan al registrarse y verifica las credenciales al iniciar sesión. Si todo es correcto, responde a la vista para mostrar el resultado al usuario.

Integración en main.py

- main.py es el punto de inicio de la aplicación. Inicializa la base de datos, crea la ventana de login y ejecuta el bucle principal de la aplicación.

Flujo de la aplicación:

Inicio de la Aplicación (main.py):

La aplicación comienza en main.py. Aquí se configura el entorno inicial:

Se crea la conexión a la base de datos, se inicializan las tablas necesarias y se cargan datos de ejemplo usando inicializacion_db.py.

```
# Inicializamos la base de datos llamando al método init_db desde el módulo inicializacion_db
init_db.init_db()
```

Luego, se instancia la ventana de inicio de sesión (LoginWindow), y se muestra al usuario como la primera pantalla de la aplicación.

```
# Creamos una instancia de LoginWindow (la ventana de inicio de sesión)
login_window = LoginWindow()

# Mostramos la ventana de login
login_window.show()
```

Ventana de Inicio de Sesión (login_window.py)

En la ventana de login, el usuario puede:

Iniciar sesión: Ingresa su correo y contraseña, y hace clic en el botón de login.

Registrarse: Hace clic en el botón de registro si aún no tiene una cuenta.

Al hacer clic en cualquiera de estos botones, la vista ejecuta una función específica:

Si el usuario hace clic en “Iniciar sesión”, el método on_button_login_clicked verifica las credenciales ingresadas llamando al controlador.



```

@Slot()
def on_button_login_clicked(self):
    print("Hemos pulsado el botón de login")
    name = self.ui.texto_usuario_correo.text()
    password = self.ui.texto_contraseña.text()

    # Comprobamos de que el usuario y la contraseña existen
    if self.usuario_controller.verificar_usuario(name, password) is not None:
        print(f"Bienvenido {name}")

    # Crear el cuadro de diálogo de bienvenida
    mensaje_bienvenida = QMessageBox(self)
    mensaje_bienvenida.setWindowTitle("Inicio de sesión exitoso")
    mensaje_bienvenida.setText(f"Bienvenido, {name}!")
    mensaje_bienvenida.setIcon(QMessageBox.Information)
    mensaje_bienvenida.exec()
    else:
        print("Credenciales incorrectas")
        mensaje_error = QMessageBox(self)
        mensaje_error.setWindowTitle("Error inicio de sesión")
        mensaje_error.setText("Credenciales incorrecta")
        mensaje_error.setIcon(QMessageBox.Critical)
        mensaje_error.exec()

```

Si el usuario hace clic en “Registrarse”, el método `on_button_crear_cuenta_clicked` oculta la ventana de login y abre la ventana de registro (RegistroWindow).



```

@Slot()
def on_button_crear_cuenta_clicked(self):
    print("Hemos pulsado el botón de registro")
    self.hide()

    if self.Ui_Registro_Equipo04 is None:
        # Creamos la ventana de registro:
        self.Ui_Registro_Equipo04 = RegistroWindow(parent=self)

    # Mostramos la ventana de registro
    self.Ui_Registro_Equipo04.show();
    # on_button_crear_cuenta_clicked

```


Validación de Credenciales (en el Controlador)

Si el usuario intenta iniciar sesión, LoginWindow llama al controlador UsuarioController.

En usuario_controller.py, el método verificar_usuario recibe los datos del usuario y, a través del modelo (CRUDUsuario en crud_usuario.py), consulta la base de datos para comprobar que el correo y la contraseña sean correctos.

```
def verificar_usuario(self, email, password):
    """
    Método para verificar si un usuario existe en la base de datos y las credenciales son correctas.

    :param email: Correo electrónico del usuario.
    :param password: Contraseña del usuario.
    :return: Un objeto Usuario si las credenciales son correctas, o None si son incorrectas.
    """
    usuario = self.crud_usuario.obtener_usuario(email) # Obtener el usuario de la base de datos por el email
    if usuario is not None and usuario.get_password == password: # Comparar si el usuario existe y la contraseña es correcta
        print("Usuario verificado correctamente.") # Mensaje de éxito si el usuario fue verificado
        return usuario # Retornar el objeto Usuario si las credenciales son correctas
    else:
        print("Credenciales incorrectas.") # Mensaje de error si las credenciales no coinciden
        return None # Retornar None si las credenciales son incorrectas
# verificar_usuario

UsuarioController
```

Resultado:

Si las credenciales son válidas, el usuario es bienvenido y se muestra un mensaje de confirmación.

Si las credenciales son incorrectas, se muestra un mensaje de error en la ventana de inicio de sesión.

Ventana de Registro (registro_window.py)

Si el usuario opta por registrarse, se abre la ventana de registro donde se le solicita ingresar un nombre de usuario, correo, y dos veces la contraseña para confirmar.

Al hacer clic en “Registrarse”, el método function_registro comprueba que las contraseñas coincidan antes de enviar los datos al controlador.

The image shows a portion of a software application's registration window. It features a dark blue header bar with the text 'Repetir contraseña' in white. Below this is a white rectangular input field for the password. At the bottom of the visible area is a prominent orange button with the text 'Regístrate' in white. The background of the window is a solid blue color.

```

@Slot()
def function_registro(self):

    nombre_usuario = self.ui.edit_usuario.text()
    email = self.ui.edit_correo.text()
    password = self.ui.edit_contrasenna.text()
    password_confirmada = self.ui.edit_r_contrasenna.text()

    if password != password_confirmada:
        print("Las contraseñas no coinciden")
        return

    print("Valores que se pasan a registrar_usuario:", email, nombre_usuario, password, password_confirmada)

    if self.usuario_controller.registrar_usuario(email, nombre_usuario, password, password_confirmada):
        print("usuario registrado exitosamente")
        mensaje_bienvenida = QMessageBox(self)
        mensaje_bienvenida.setWindowTitle("Registro exitoso")
        mensaje_bienvenida.setText("Usuario creado correctamente")
        mensaje_bienvenida.setIcon(QMessageBox.Information)
        mensaje_bienvenida.exec()
        self.vaciarCamposDeTexto()
    else:
        print("Error al crear el usuario")
        mensaje_error = QMessageBox(self)
        mensaje_error.setWindowTitle("Error Registro")
        mensaje_error.setText("Error al crear el usuario")
        mensaje_error.setIcon(QMessageBox.Critical)
        mensaje_error.exec()

```

Proceso de Registro (en el Controlador)

En UsuarioController, el método registrar_usuario valida los datos ingresados por el usuario y verifica que el correo no esté ya registrado en la base de datos.

```

def verificar_usuario(self, email, password):
    """
    Método para verificar si un usuario existe en la base de datos y las credenciales son correctas.

    :param email: Correo electrónico del usuario.
    :param password: Contraseña del usuario.
    :return: Un objeto Usuario si las credenciales son correctas, o None si son incorrectas.
    """
    usuario = self.crud_usuario.obtener_usuario(email) # Obtener el usuario de la base de datos por el email
    if usuario is not None and usuario.get_password == password: # Comparar si el usuario existe y la contraseña es correcta
        print("Usuario verificado correctamente.") # Mensaje de éxito si el usuario fue verificado
        return usuario # Retornar el objeto Usuario si las credenciales son correctas
    else:
        print("Credenciales incorrectas.") # Mensaje de error si las credenciales no coinciden
        return None # Retornar None si las credenciales son incorrectas
# verificar_usuario

```

Si no existe un usuario con el correo ingresado, se envían los datos al modelo (CRUDUsuario) para crear un nuevo usuario en la base de datos.

Resultado:

Si el registro es exitoso, se muestra un mensaje de confirmación y se limpia el formulario.

Si ocurre un error (como que el correo ya exista), se notifica al usuario en la ventana de registro.

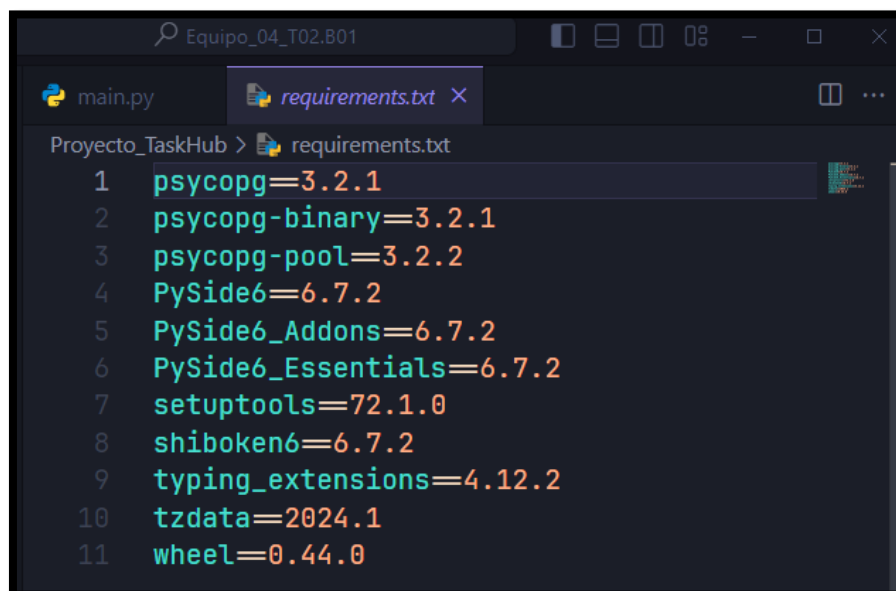
Finalización

Tras completar el registro o iniciar sesión, el flujo inicial de la aplicación termina, y el usuario está listo para interactuar con el resto de funcionalidades que se desarrollen más adelante en la aplicación.

Despliegue de la aplicación

Como crear el entorno virtual en el proyecto

Para crear el entorno virtual, primero tenemos que tener un archivo requirements, este archivo contiene todas las dependencias que nos va a hacer falta en nuestro proyecto, en nuestro caso, nuestro archivo tiene las siguientes dependencias:

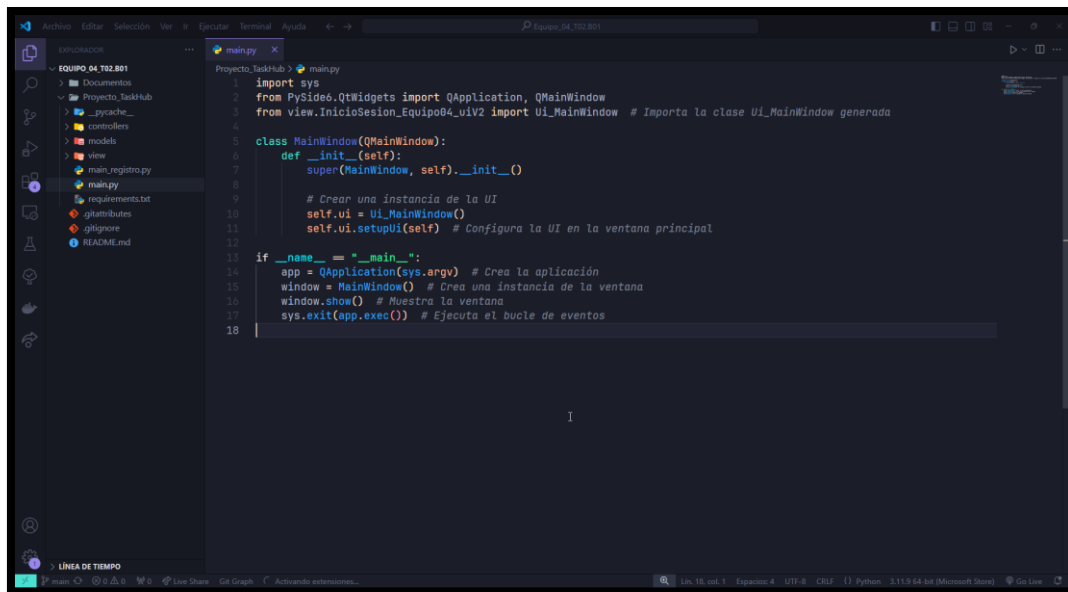


```
Equipo_04_T02.B01
main.py requirements.txt
Proyecto_TaskHub > requirements.txt
1  psycpg==3.2.1
2  psycpg-binary==3.2.1
3  psycpg-pool==3.2.2
4  PySide6==6.7.2
5  PySide6_Addons==6.7.2
6  PySide6_Essentials==6.7.2
7  setuptools==72.1.0
8  shiboken6==6.7.2
9  typing_extensions==4.12.2
10 tzdata==2024.1
11 wheel==0.44.0
```

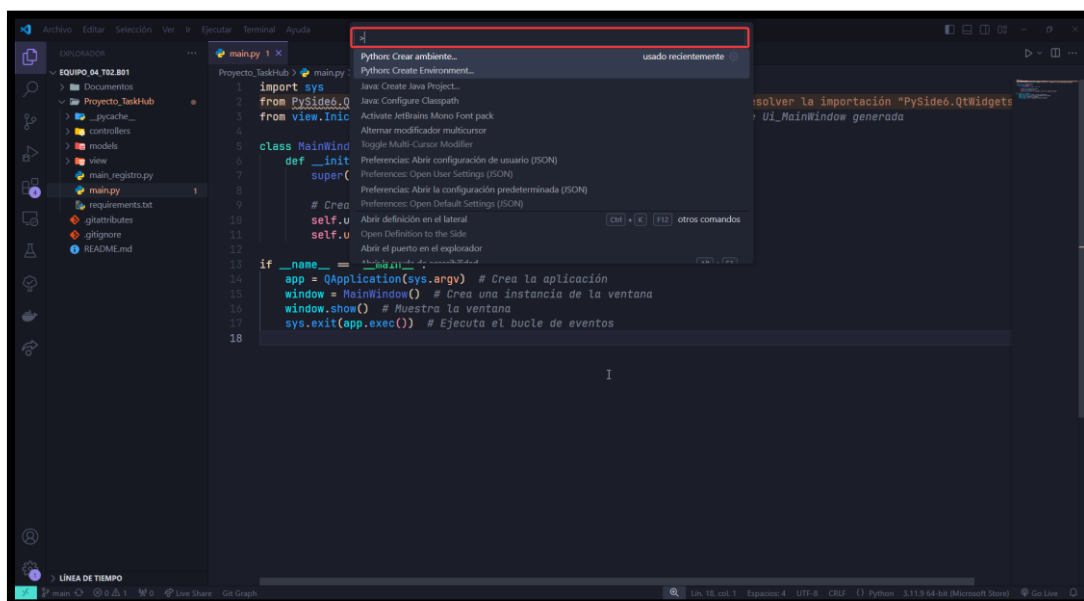
Una vez que tenemos nuestro archivo de dependencias localizado, podemos proceder a instalar nuestro entorno virtual.

Crear entorno de trabajo en Python:

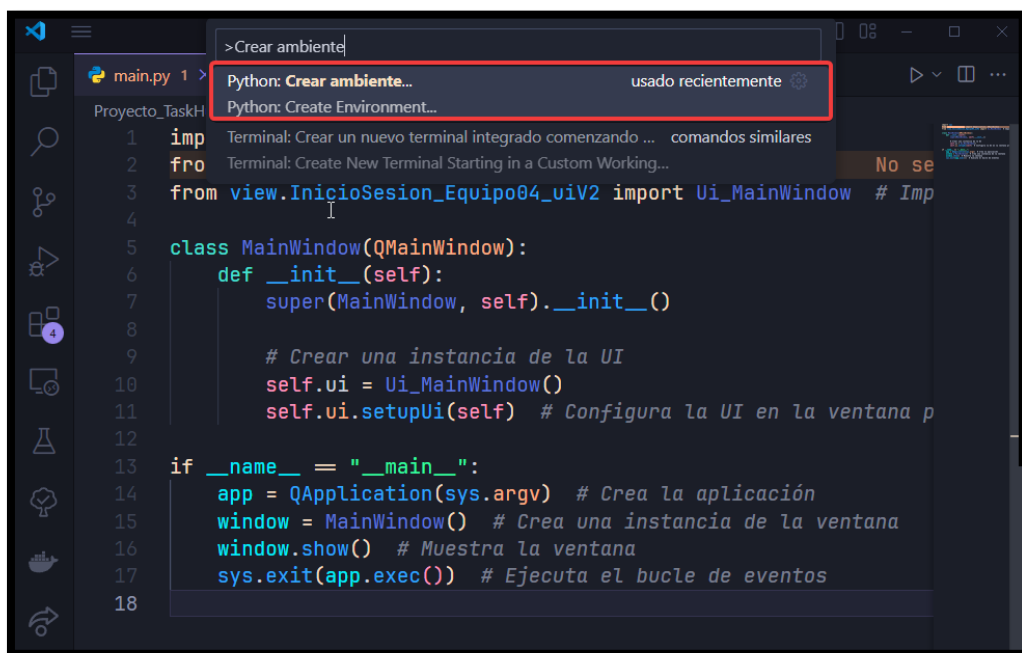
Lo primero que tenemos que hacer es ir a nuestro entorno de trabajo, en nuestro caso visual studio code:



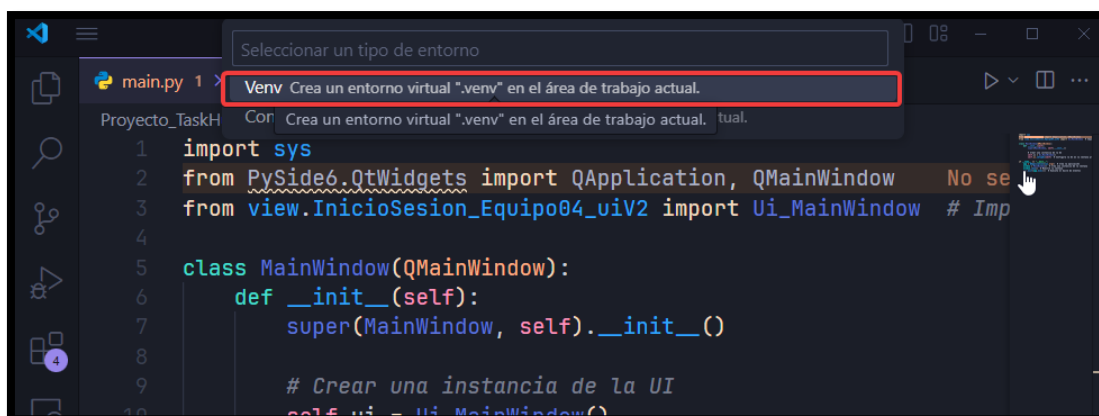
Una vez aquí, pulsaremos sobre la tecla F1 y se nos abrirá una barra en la que podremos buscar acciones:



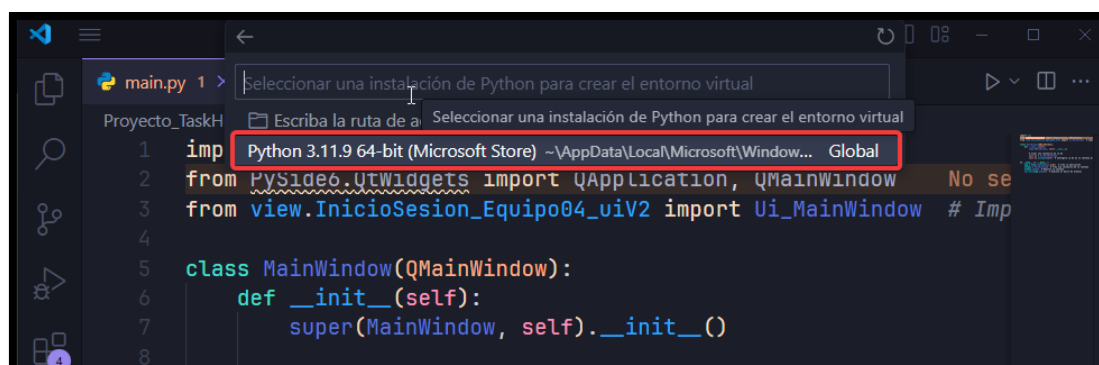
Una vez aquí buscaremos crear ambiente y pulsaremos sobre este:



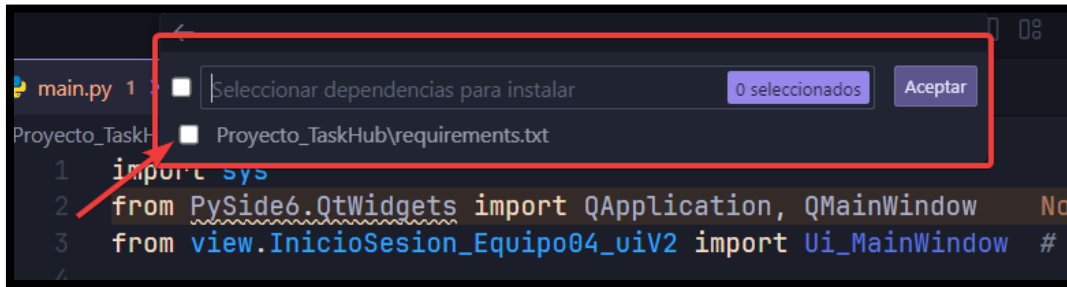
Ahora damos en Venv



Ahora tendremos que seleccionar nuestra versión de Python correspondiente, en este caso como solo tengo una versión instalada marcaremos esta

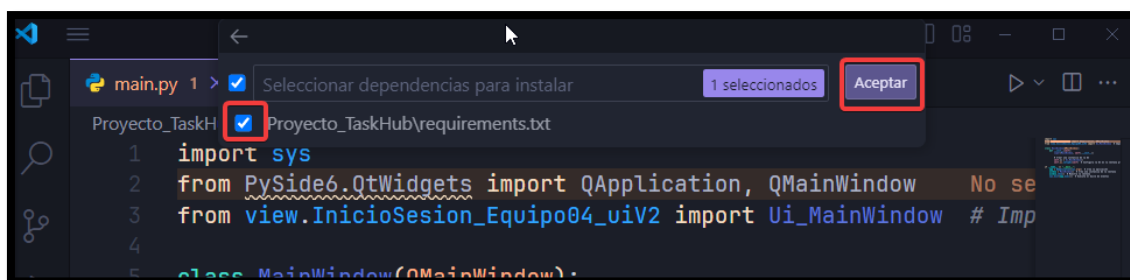


Ahora tendremos que marcar los requerimientos que nos hacen falta para nuestro proyecto, que aquí te instala todos los paquetes que necesites

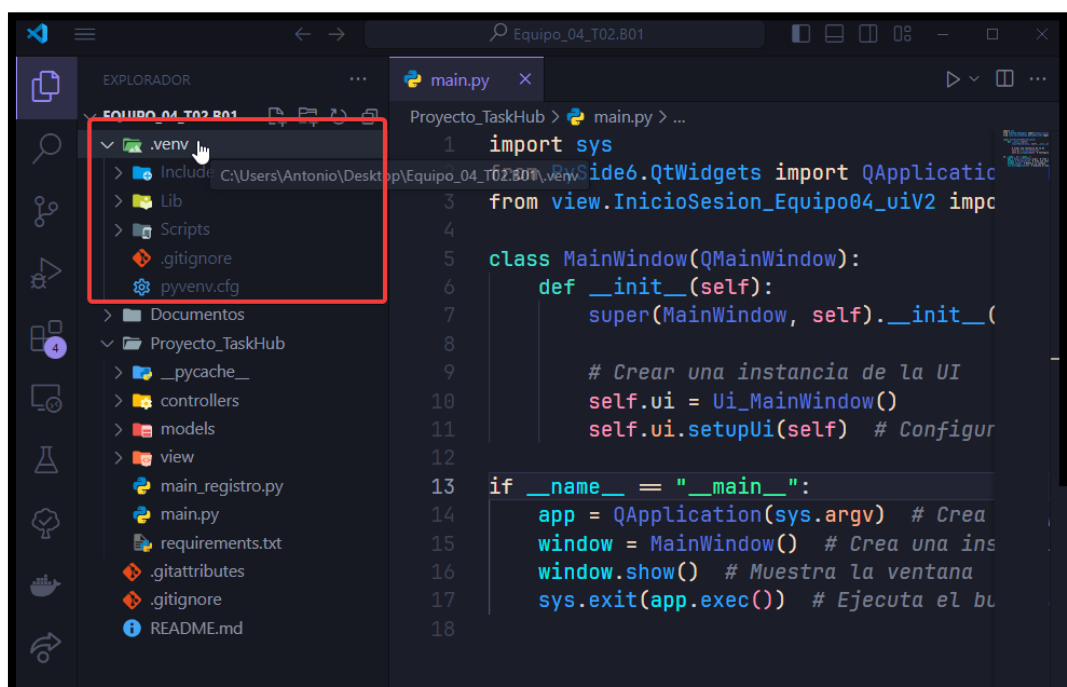


Este archivo es configurado por el usuario al cual le pertenece el proyecto, es muy útil ya que con esto no tenemos que pasar el entorno virtual ya que se descarga todos los requisitos

Una vez seleccionado damos en aceptar:



Y una vez hecho esto, se nos empezara a crear automáticamente nuestro entorno:



Como podemos ver ya tenemos nuestro entorno creado correctamente y las librerías nos las coge sin ningún error.

Como hacerlo por comandos

python -m venv "nombre_del_entorno"

Como activarlo:

Para activar el entorno podemos usar el siguiente comando: `.\venv\Scripts\activate`

Una vez activado nos saldrá al principio de la ruta del archivo (.venv) en verde.

```
PS C:\Users\david\Documents\GitHub\Equipo_04_T02.B01> .\venv\Scripts\activate
(.venv) PS C:\Users\david\Documents\GitHub\Equipo_04_T02.B01> 
```

Que es el archivo requirements.txt y qué contiene el vuestro en concreto (explicar un poco que dependencias hay dentro)

El archivo requirements.txt es un archivo de texto plano que enumera todas las dependencias de tu proyecto Python junto con las versiones específicas de cada paquete.

Bibliografía:

Maldonado, D. (2024, abril 22). Archivo requirements.txt ¿Cómo crearlo? Daniel Maldonado.

<https://danielmaldonado.com.ar/python/archivo-requirements-txt-como-crearlo/>

Que dependencias tiene el archivo requirements.txt:

psycopg==3.2.1: Este es el adaptador de PostgreSQL para Python. Permite interactuar con bases de datos PostgreSQL desde Python, utilizando el protocolo nativo de PostgreSQL.

psycopg-binary==3.2.1: Es una versión precompilada de psycopg que incluye todas las dependencias en un solo paquete. Esto facilita la instalación, especialmente cuando se desea evitar problemas de compilación en diferentes entornos.

psycopg-pool==3.2.2: Proporciona funcionalidades de "pooling" para conexiones de PostgreSQL. Un pool de conexiones mejora el rendimiento al reutilizar conexiones abiertas en lugar de abrir y cerrar conexiones para cada operación.

PySide6==6.7.2: Es una biblioteca para construir interfaces gráficas (GUI) en Python basada en Qt6. Ofrece una amplia variedad de componentes para crear aplicaciones visuales complejas.

PySide6_Addons==6.7.2: Incluye módulos adicionales para PySide6, que amplían las capacidades de la interfaz gráfica con funcionalidades avanzadas y complementarias no incluidas en el paquete principal.

PySide6_Essentials==6.7.2: Contiene módulos esenciales para el desarrollo de aplicaciones con PySide6, enfocándose en las herramientas y widgets básicos necesarios para crear interfaces de usuario.

setuptools==72.1.0: Es una herramienta que facilita el empaquetado y la distribución de proyectos Python, especialmente para gestionar sus dependencias y hacerlos instalables mediante pip.

shiboken6==6.7.2: Es el generador de enlaces (bindings) para PySide6, permitiendo que el código de Qt6 se conecte con Python. Esto es esencial para que las aplicaciones GUI de PySide6 funcionen correctamente.

typing_extensions==4.12.2: Proporciona compatibilidad con características de tipado avanzadas en Python, útiles en versiones de Python que no incluyen algunos de estos tipos de forma nativa.

tzdata==2024.1: Contiene información sobre zonas horarias de todo el mundo, esencial para aplicaciones que deben trabajar con fechas y horas en diferentes regiones.

wheel==0.44.0: Es una herramienta para construir y descomprimir paquetes Python en el formato wheel, que es el estándar moderno para distribuir archivos binarios de Python, acelerando la instalación de paquetes.

Como instalar las dependencias desde el archivo requirements.txt

Para instalar las dependencias desde el archivo requirements.txt ejecutaremos el comando: `pip install -r .\requirements.txt`

```
(.venv) PS C:\Users\pedro\Desktop\Equipo_04_T02.001> cd .\Proyecto_TaskHub\
(.venv) PS C:\Users\pedro\Desktop\Equipo_04_T02.001\Proyecto_TaskHub> pip install -r .\requirements.txt
Requirement already satisfied: psycpg==3.2.1 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 1)) (3.2.1)
Requirement already satisfied: psycpg-binary==3.2.1 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 2)) (3.2.1)
Requirement already satisfied: psycpg-pool==3.2.2 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 3)) (3.2.2)
Requirement already satisfied: PySide6==6.7.2 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 4)) (6.7.2)
Requirement already satisfied: PySide6_Addons==6.7.2 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 5)) (6.7.2)
Requirement already satisfied: PySide6_Essentials==6.7.2 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 6)) (6.7.2)
Requirement already satisfied: setuptools==72.1.0 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 7)) (72.1.0)
Requirement already satisfied: shiboken6==6.7.2 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 8)) (6.7.2)
Requirement already satisfied: typing_extensions==4.12.2 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 9)) (4.12.2)
Requirement already satisfied: tzdata==2024.1 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 10)) (2024.1)
Requirement already satisfied: wheel==0.44.0 in c:\users\pedro\desktop\equipo_04_t02.001\.venv\lib\site-packages (from -r .\requirements.txt (line 11)) (0.44.0)
(.venv) PS C:\Users\pedro\Desktop\Equipo_04_T02.001\Proyecto_TaskHub>
```

CONVERTIR ARCHIVO .UI EN .PY

Para convertir el archivo .ui a .py utilizaremos el siguiente comando:

`pyside6-uic archivo.ui -o archivo.py` un ejemplo de este comando sería este:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
(.venv) PS C:\Users\pedro\Desktop\Equipo_04_T02.001\Proyecto_TaskHub> pyside6-uic Views/qt/InicioSesion_Equipo04_V2.ui -o Views/qt/qt_inicio_sesion.py
```


DOCKER Y BBDD POSTGRESQL:

Mostrar como levantar el servicio de docker y que debe estar activo antes de continuar (esto en Windows es tan sencillo como abrir docker desktop)

Antes de usar el comando Docker-compose up –build es importante que tengamos abierto el docker desktop. Este comando coge los datos del archivo Docker-compose.yml y crea un contenedor en Docker desktop.

Para crear el contenedor de docker con la base de datos y el usuario, hemos creado un **docker-compose**, el cual tiene el siguiente contenido:

Aquí a destacar es que el nombre del contenedor se llama equipo eq_04_taskHub_db y el nombre de la base de datos igual que el nombre del contenedor, el usuario es admin y la contraseña es 0000

```
services:
  db:
    image: postgres:latest # Última versión de la imagen oficial de PostgreSQL
    container_name: eq_04_taskHub_db # Nombre personalizado del contenedor
    environment:
      POSTGRES_DB: eq_04_taskHub_db # Nombre de la base de datos a crear
      POSTGRES_USER: admin # Nombre del usuario administrador
      POSTGRES_PASSWORD: "0000" # Contraseña del usuario administrador
      PGDATA: /var/lib/postgresql/data/pgdata # Cambiar el directorio de datos
    stdin_open: true # Equivalente a `~it`
    tty: true # Habilita el pseudo-terminal interactivo
    ports:
      - "54320:5432" # Mapeo del puerto 54320 del host al puerto 5432 del contenedor
    volumes:
      - ./postgres-data:/var/lib/postgresql/data/pgdata # Montaje del volumen para persistir datos
    restart: always # Configura el contenedor para que se reinicie siempre que se detenga

volumes:
  postgres-data:
    driver: local # Especifica que el volumen se almacenará localmente
```

Una vez tengamos esto tendremos que ejecutar por el cmd el siguiente comando: Docker-compose up –build y de este modo se nos creara nuestro contenedor con los datos correspondiente para poder acceder a nuestra base de datos, el comando en la terminal tenemos que estar en el mismo lugar que donde tenemos nuestro archivo. yml

Microsoft Windows [Versión 10.0.22631.4317]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\anton\Desktop\Equipo_04_T02.B01>dir
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: E841-5878

Directorio de C:\Users\anton\Desktop\Equipo_04_T02.B01

| Fecha y hora | Nombre | Tamaño |
|------------------|--------------------------|------------------------------|
| 29/10/2024 17:41 | <DIR> | |
| 29/10/2024 17:51 | .. | |
| 29/10/2024 17:31 | 68 .gitattributes | |
| 29/10/2024 17:36 | <DIR> | |
| 29/10/2024 17:29 | 1.107 docker-compose.yml | |
| 29/10/2024 17:31 | <DIR> | |
| 29/10/2024 17:40 | <DIR> | |
| 29/10/2024 17:31 | 556 README.md | |
| 29/10/2024 17:31 | 3 archivos | 1.731 bytes |
| 29/10/2024 17:31 | 5 dirs | 284.286.922.752 bytes libres |

C:\Users\anton\Desktop\Equipo_04_T02.B01>docker-compose up --build

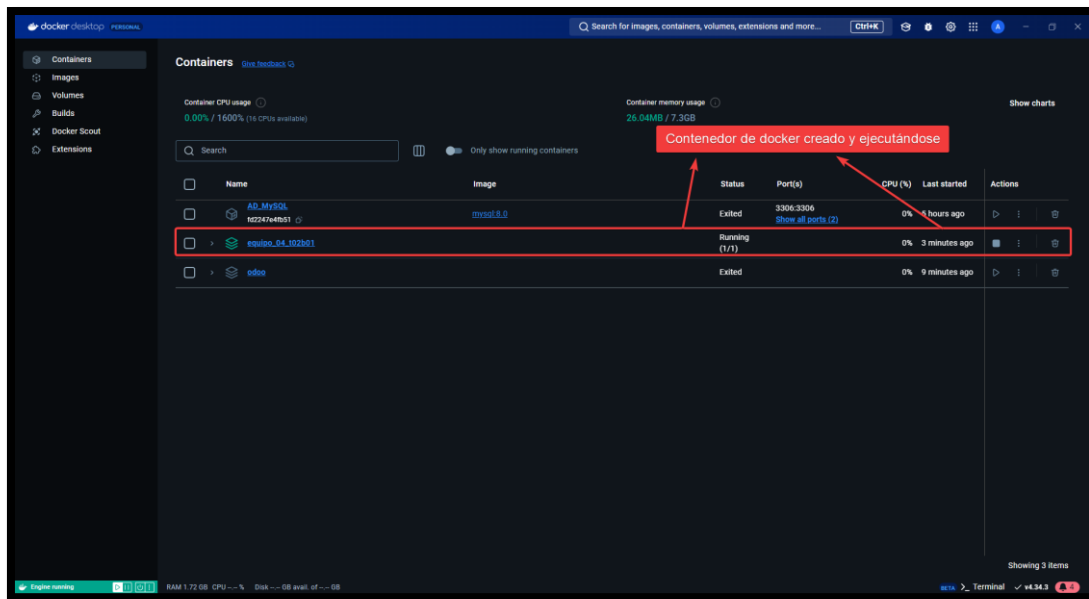
Muestro que tenemos el archivo en la ruta actual

Comando para crear el contenedor de base de datos

```
Microsoft Windows [Versión 10.0.22631.4317]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\anton\Desktop\Equipo_04_T02.B01>docker-compose up --build
Time="2024-10-29T18:28:43+01:00" level=warning msg="C:\\Users\\anton\\Desktop\\Equipo_04_T02.B01\\docker-compose.yml: the attribute 'version' is obsolete, it will be ignored, please remove it to avoid potential confusion"
[+] Running 2/2
  ✔ network equipo_04_t02b01_default Created 0.1s
  ✔ Container eq0taskhubdb Created 0.1s
Attaching to eq0taskhubdb
eq0taskhubdb: The files belonging to this database system will be owned by user "postgres".
eq0taskhubdb: This user must also own the server process.
eq0taskhubdb:
eq0taskhubdb: The database cluster will be initialized with locale "en_US.utf8".
eq0taskhubdb: The default database encoding has accordingly been set to "UTF8".
eq0taskhubdb: The default text search configuration will be set to "english".
eq0taskhubdb:
eq0taskhubdb: Data page checksums are disabled.
eq0taskhubdb:
eq0taskhubdb: fixing permissions on existing directory /var/lib/postgresql/data/pgdata ... ok
eq0taskhubdb: creating subdirectories ... ok
eq0taskhubdb: selecting dynamic shared memory implementation ... posix
eq0taskhubdb: selecting default "max_connections" ... 100
eq0taskhubdb: selecting default "shared_buffers" ... 128MB
eq0taskhubdb: selecting default time zone ... Etc/UTC
eq0taskhubdb: creating configuration files ... ok
eq0taskhubdb: running bootstrap script ... ok
eq0taskhubdb: performing post-bootstrap initialization ... ok
eq0taskhubdb: syncing data to disk ... ok
eq0taskhubdb:
initdb: warning: enabling "trust" authentication for local connections
initdb: hint: You can change this by editing pg_hba.conf or using the option -A, or --auth-local and --auth-host, the next time you run initdb.
Success. You can now start the database server using:
    pg_ctl -D /var/lib/postgresql/data/pgdata -l logfile start
waiting for server to start....2024-10-29 17:28:55.036 UTC [49] LOG: starting PostgreSQL 17.0 (Debian 17.0-1.pgd12b1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-b
2024-10-29 17:28:55.038 UTC [49] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2024-10-29 17:28:55.060 UTC [52] LOG: database system was shut down at 2024-10-29 17:28:52 UTC
done
server started
CREATE DATABASE
/usr/local/bin/docker-entrypoint.sh: ignoring /docker-entrypoint-initdb.d/*
2024-10-29 17:28:57.619 UTC [49] LOG: received fast shutdown request
waiting for server to shut down....2024-10-29 17:28:57.624 UTC [49] LOG: aborting any active transactions
```

Como podemos ver se nos ha lanzado correctamente, por lo que esto significa que ya tenemos nuestra base de datos creada:



Explicar que es docker-compose y para que se utiliza.

Docker Compose es una herramienta para definir y ejecutar aplicaciones de Docker de varios contenedores. En Compose, se usa un archivo YAML para configurar los servicios de la aplicación. Después, con un solo comando, se crean y se inician todos los servicios de la configuración.

Bibliografía:

aahill. (s. f.). Uso de Docker Compose para implementar varios contenedores. Microsoft.com. Recuperado 3 de noviembre de 2024, de <https://learn.microsoft.com/es-es/azure/ai-services/containers/docker-compose-recipe>

Explicar el contenido de dicho archivo para que se entienda.

services: Esta sección agrupa los servicios que la aplicación necesita. En este caso, solo hay un servicio llamado db.

Servicio db:

image: postgres: Usa la última versión de la imagen oficial de PostgreSQL.

container_name: eq_04_taskHub_db: Especifica un nombre personalizado para el contenedor, facilitando su identificación.

environment: Define variables de entorno necesarias para configurar la base de datos PostgreSQL.

POSTGRES_DB: eq_04_taskHub_db: Nombre de la base de datos que se creará al iniciar el contenedor.

POSTGRES_USER: admin: Usuario administrador de la base de datos.

POSTGRES_PASSWORD: "0000": Contraseña para el usuario administrador.

PGDATA: /var/lib/postgresql/data/pgdata: Especifica el directorio donde se guardarán los datos de PostgreSQL.

stdin_open: true y tty: true: Permiten una sesión interactiva con el contenedor, útil para tareas de depuración.

ports:

"54320:5432": Mapea el puerto 5432 del contenedor (puerto predeterminado de PostgreSQL) al puerto 54320 en el host. Esto permite acceder a la base de datos desde el host en el puerto 54320.

volumes:

./postgres-data:/var/lib/postgresql/data/pgdata: Monta un volumen local para persistir los datos en el host. Así, los datos no se perderán si el contenedor se reinicia o elimina.

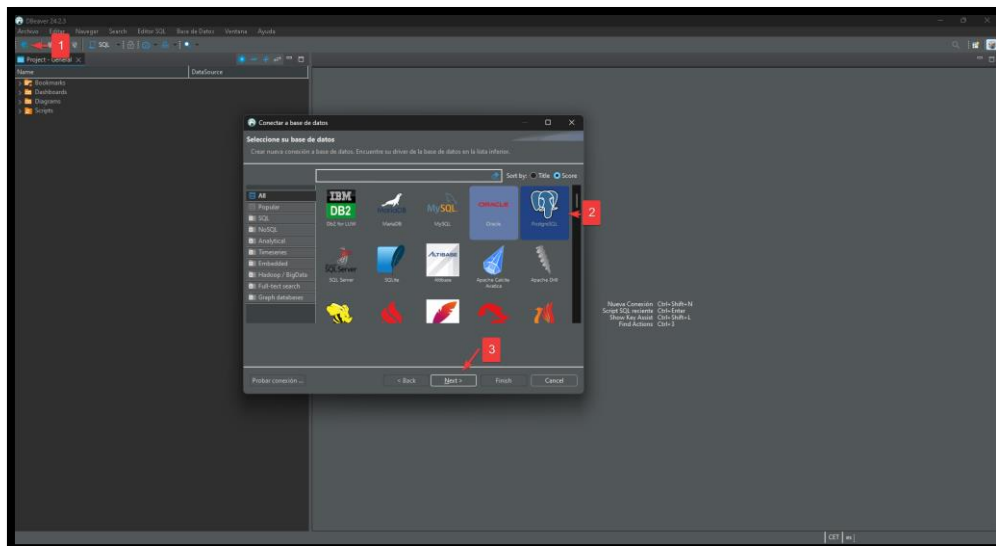
restart: always: Configura el contenedor para que se reinicie automáticamente en caso de fallos o si se reinicia el sistema.

volumes: Define un volumen llamado postgres-data:

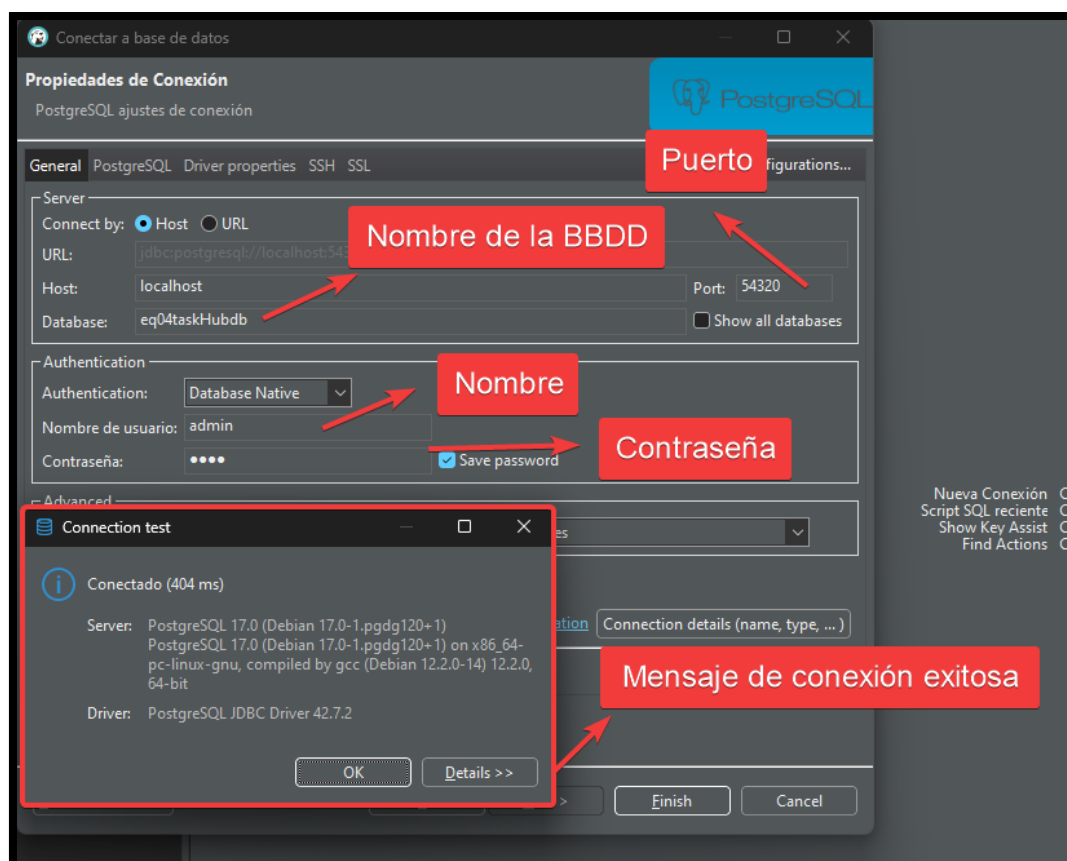
driver: local: Indica que el volumen se almacenará localmente, en el sistema de archivos del host.

Explicar cómo conectarnos a la bbdd con algún software de gestión de bbdd (podéis usar DBeaver u otro)

Para esto abrimos la aplicación y creamos una nueva sesión:



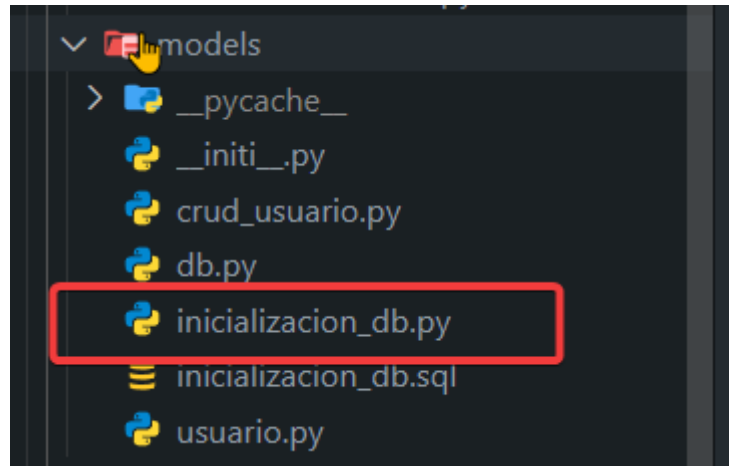
Una vez dado a la base de datos que nos queremos conectar, pondremos nuestros datos correspondientes:



Una vez dado ha ok pulsamos sobre finish y con esto ya estaríamos dentro de nuestra BBDD

En PYTHON:

Tener un archivo “inicialización_db.py” (será parte del directorio “models”). Este archivo contendrá el código que permite inicializar y popular la bbdd sin que el usuario que use la aplicación tenga que hacer nada para usar la aplicación.



Este archivo tenéis que usarlo en vuestro código para que cuando arranque la aplicación, sea llamado para crear y popular la bbdd.

```
import sys
from PySide6.QtWidgets import QApplication
from views.login_window import LoginWindow # Importa la clase Ui_MainWindow generada
from models import inicializacion_db as init_db

if __name__ == "__main__":
    # Creamos una instancia de QApplication, pasándole los argumentos del sistema
    app = QApplication(sys.argv)

    # Inicializamos la base de datos llamando al método init_db desde el módulo inicializacion_db
    init_db.init_db()

    # Creamos una instancia de LoginWindow (la ventana de inicio de sesión)
    login_window = LoginWindow()

    # Mostramos la ventana de login
    login_window.show()

    # Ejecutamos el bucle de eventos de la aplicación para esperar interacciones del usuario
    sys.exit(app.exec())

# __main__
```