

**IES Carrillo Curso 24/25**

**Desarrollo de aplicaciones multiplataforma**

**Equipo 04:**

**Pedro González Martín**

**Antonio Gómez Camarena**

**David Castro Soriano**

**Tema 02 – Boletín 03**

indice

# Contenido

[Contenido 1](#_Toc181532851)

[Integración de la interfaz creada con Qt Designer (archivo .ui) en Python. 3](#_Toc181532852)

[Integración de Eventos, Señales y Slots a la Interfaz en Python. 4](#_Toc181532853)

[Integración de la BBDD en la aplicación. 6](#_Toc181532854)

[Resumen sobre la Aplicación 13](#_Toc181532855)

[1. Modelo (Models) 13](#_Toc181532856)

[2. Vista (Views) 13](#_Toc181532857)

[3. Controlador (Controllers 13](#_Toc181532858)

[Integración en main.py 14](#_Toc181532859)

[Flujo de la aplicación: 14](#_Toc181532860)

[Inicio de la Aplicación (main.py): 14](#_Toc181532861)

[Ventana de Inicio de Sesión (login\_window.py) 14](#_Toc181532862)

[Validación de Credenciales (en el Controlador) 16](#_Toc181532863)

[Ventana de Registro (registro\_window.py) 16](#_Toc181532864)

[Proceso de Registro (en el Controlador) 17](#_Toc181532865)

[Finalización 18](#_Toc181532866)

[Despliegue de la aplicación 18](#_Toc181532867)

[Como crear el entorno virtual en el proyecto 18](#_Toc181532868)

[Como hacerlo por comandos 22](#_Toc181532869)

[Como activarlo: 22](#_Toc181532870)

[Que es el archivo requirements.txt y qué contiene el vuestro en concreto (explicar un poco que dependencias hay dentro) 22](#_Toc181532871)

[Como instalar las dependencias desde el archivo requirements.txt 23](#_Toc181532872)

[CONVERTIR ARCHIVO .UI EN .PY 23](#_Toc181532873)

[DOCKER Y BBDD POSTGRESQL: 24](#_Toc181532874)

[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](#_Toc181532875)

[Explicar que es docker-compose y para que se utiliza. 26](#_Toc181532876)

[Explicar el contenido de dicho archivo para que se entienda. 26](#_Toc181532877)

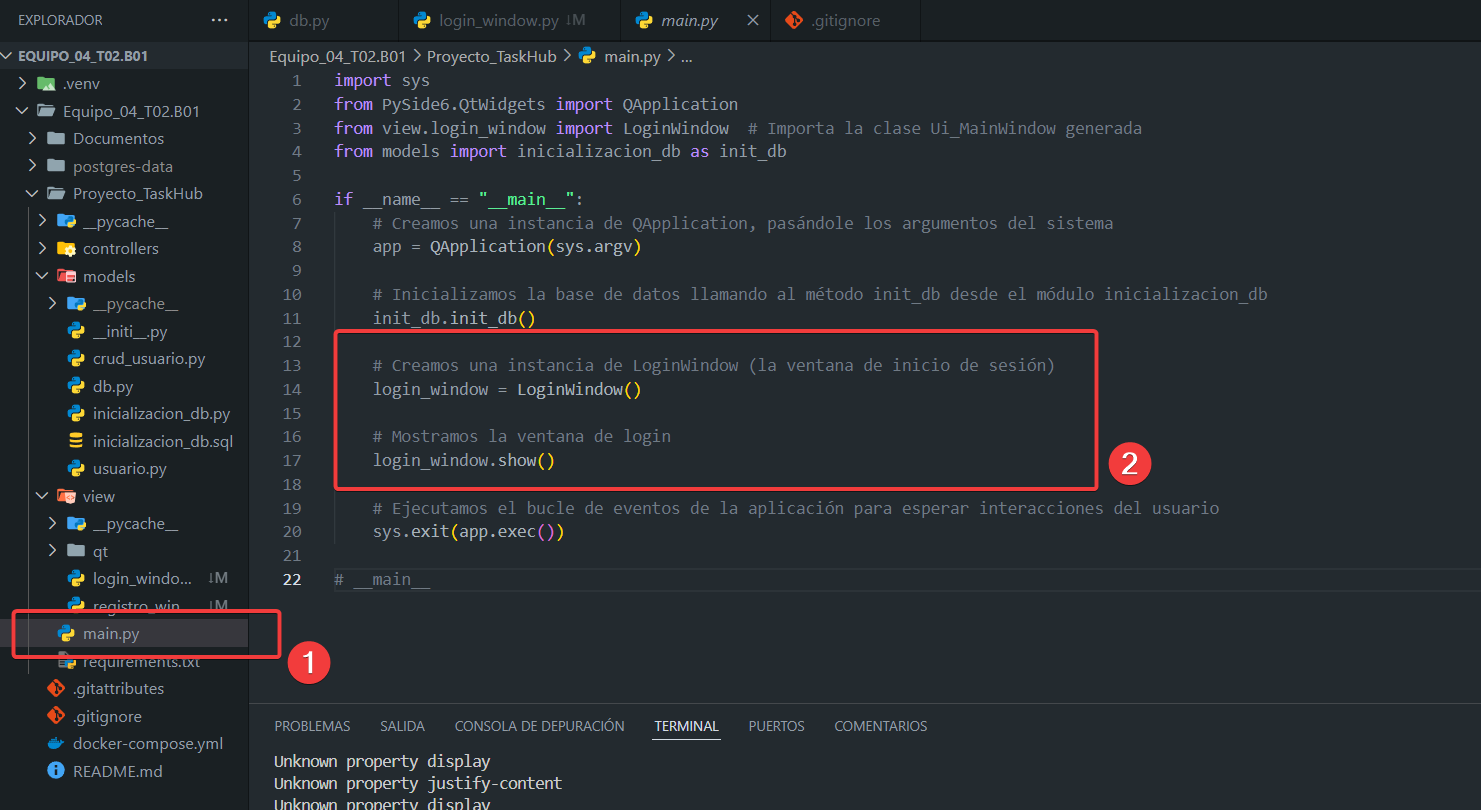
[Explicar cómo conectarnos a la bbdd con algún software de gestión de bbdd (podéis usar DBeaver u otro) 27](#_Toc181532878)

[En PYTHON: 29](#_Toc181532879)

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

Creamos los .py correspondientes a cada ventana introduciendo el comando (pyside6-uic “clase.ui” -o “clase.py”) donde clase.ui es la interfaz generada por Qtdesigner y clase.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.



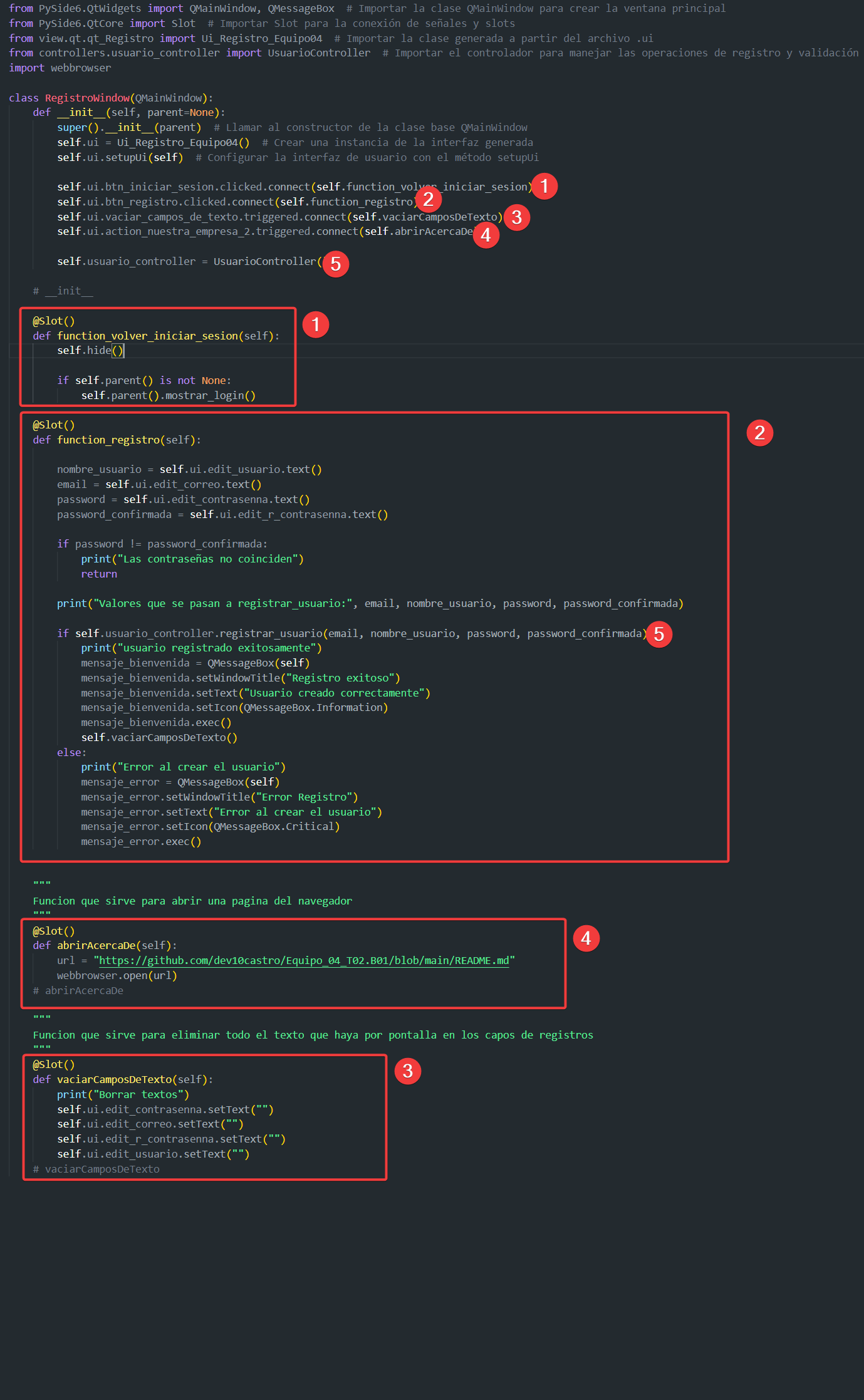
1. Clase main.
2. Hacemos visible la interfaz con “login.show()”.

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.

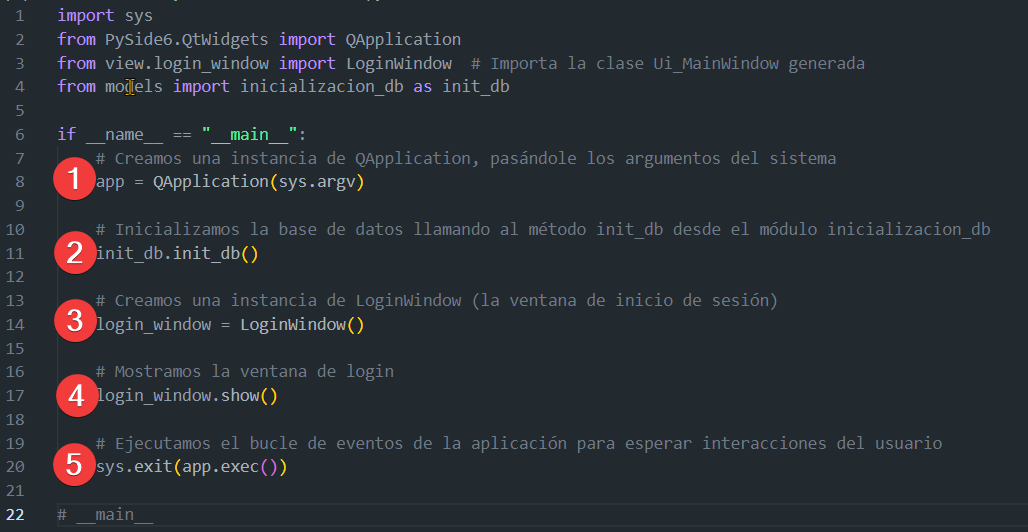
# Integración de Eventos, Señales y Slots a la Interfaz en Python.



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.

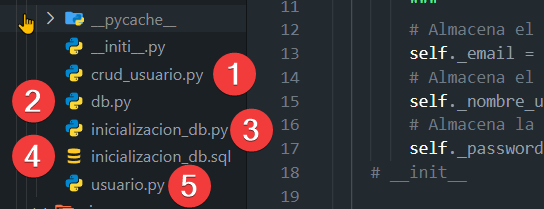


1. Función que comprueba el correo y contraseña en la base de datos y si es correcto nos muestra un QMessageBox, en caso contrario nos mostrara otro avisando de que las credenciales son incorrectas.
2. Oculta la ventana login indicamos cual es la clase registro y la mostramos.
3. Nos lleva, al pulsar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



1. Crea una instancia de Qapplication pasamdole 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.





1. 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.

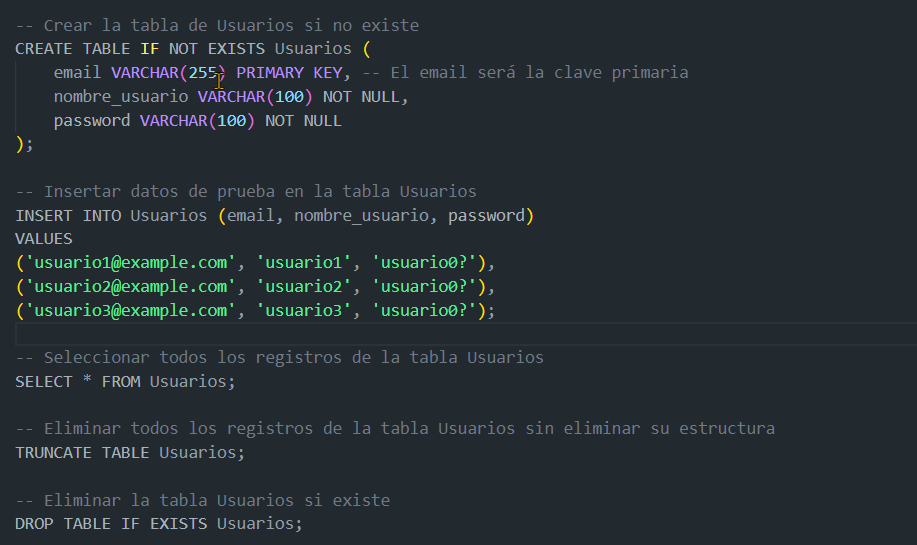


1. inicializacionbd.py: Tiene los métodos create\_tables(1), insert\_data(2) e init\_db(3), este último se encarga de llamar a los otros dos métodos para crear las tablas e insertar datos de ejemplo.





1. inicialización\_db.sql: Si la tabla usuarios no existe, la crea e inserta datos de prueba en la base de datos.



1. usuario.py: Clase objeto Usuario, con su constructor, sus getters y setter y los métodos \_\_str\_\_ y \_\_repr\_\_ .





\*\* 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

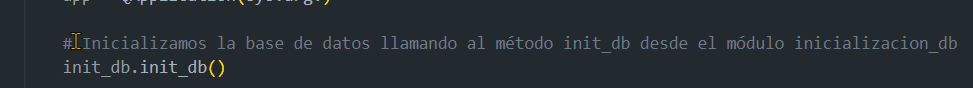
* main.py es el punto de inicio de la aplicación. Inicializa la base de datos, crea la ventana de login y ejecuta el bucle principal de la aplicación.

## Flujo de la aplicación:

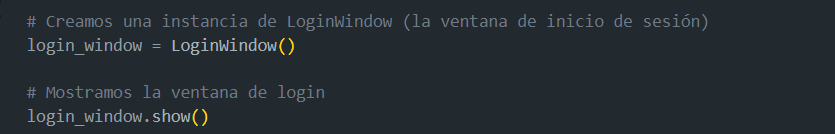
### Inicio de la Aplicación (main.py):

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.



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



### Ventana de Inicio de Sesión (login\_window.py)

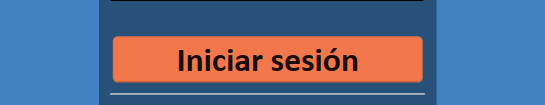
En la ventana de login, el usuario puede:

**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:

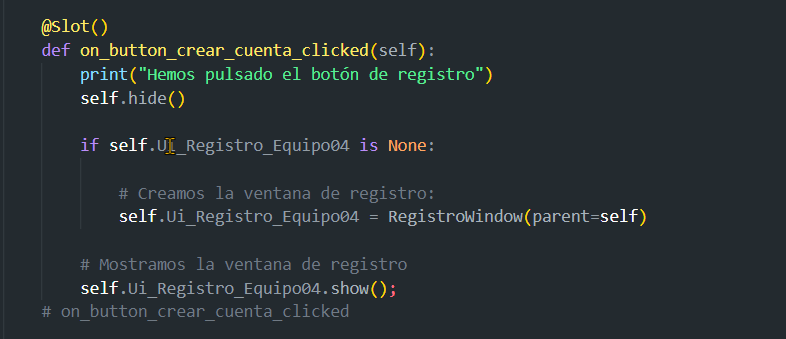
Si el usuario hace clic en “Iniciar sesión”, el método on\_button\_login\_clicked verifica las credenciales ingresadas llamando al controlador.





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).

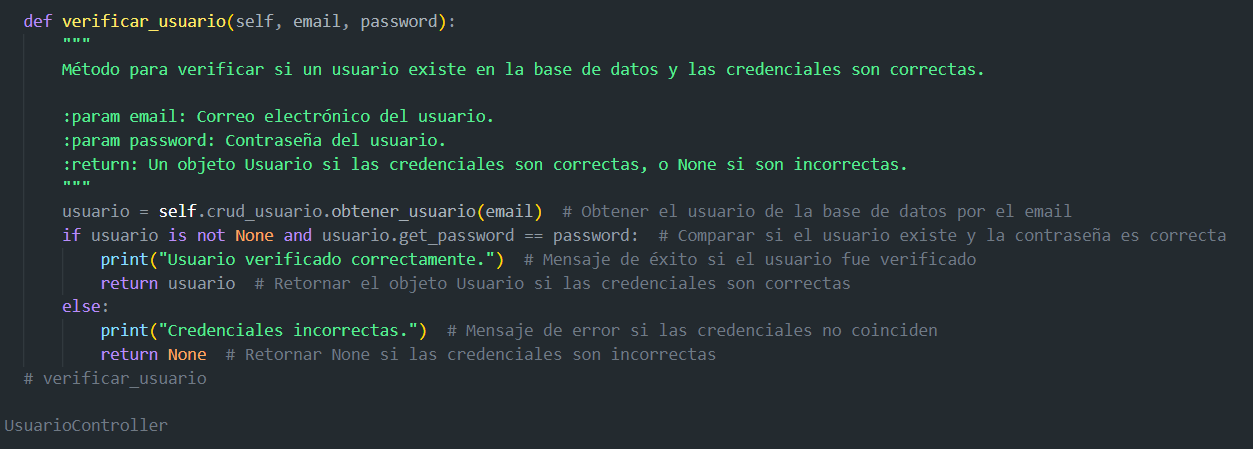




### Validación de Credenciales (en el Controlador)

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.



Resultado:

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

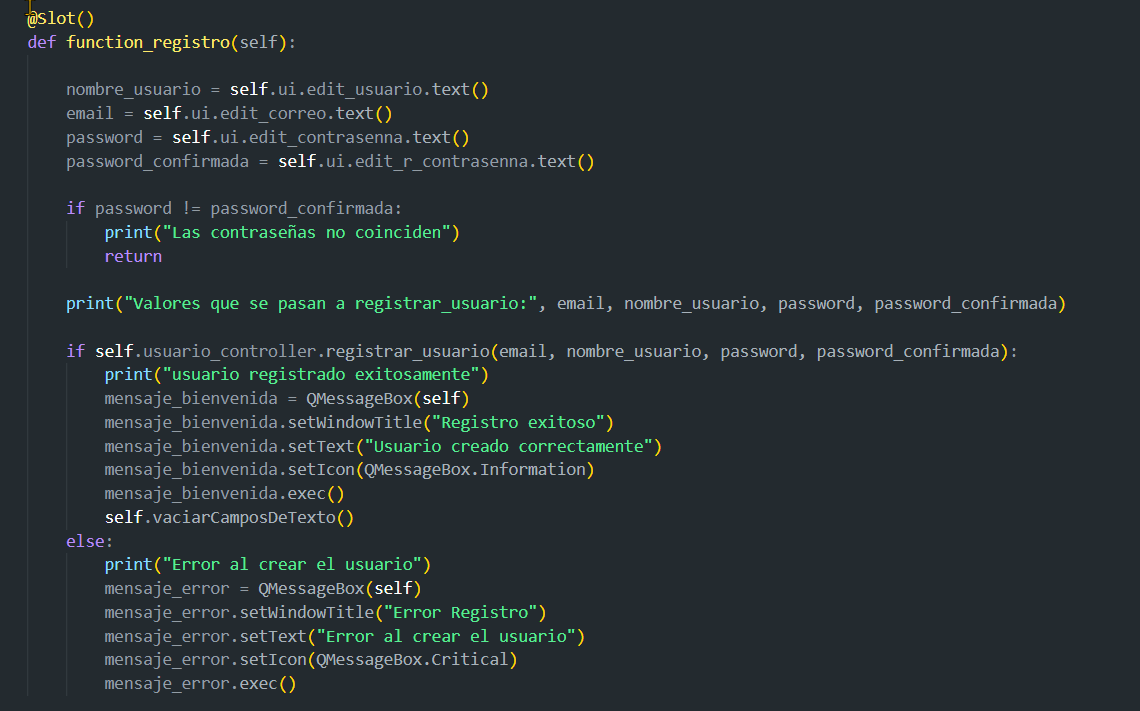
Si las credenciales son incorrectas, se muestra un mensaje de error en la ventana de inicio de sesión.

### 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.

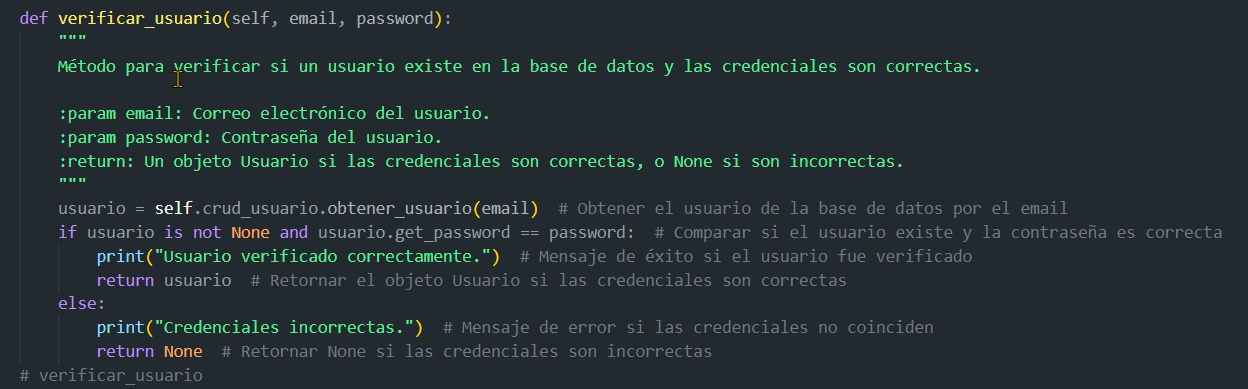
Al hacer clic en “Registrarse”, el método function\_registro comprueba que las contraseñas coincidan antes de enviar los datos al controlador.





### 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.



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.

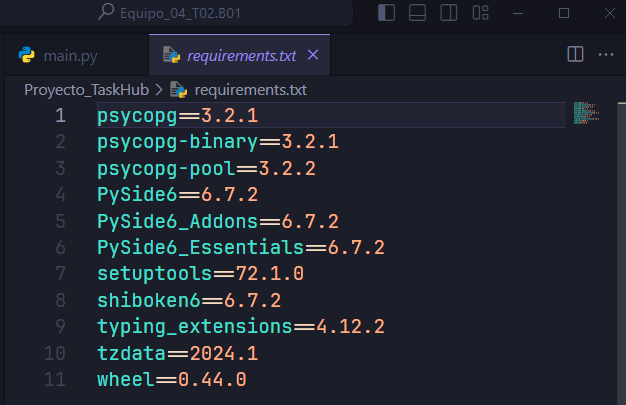
### Finalización

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

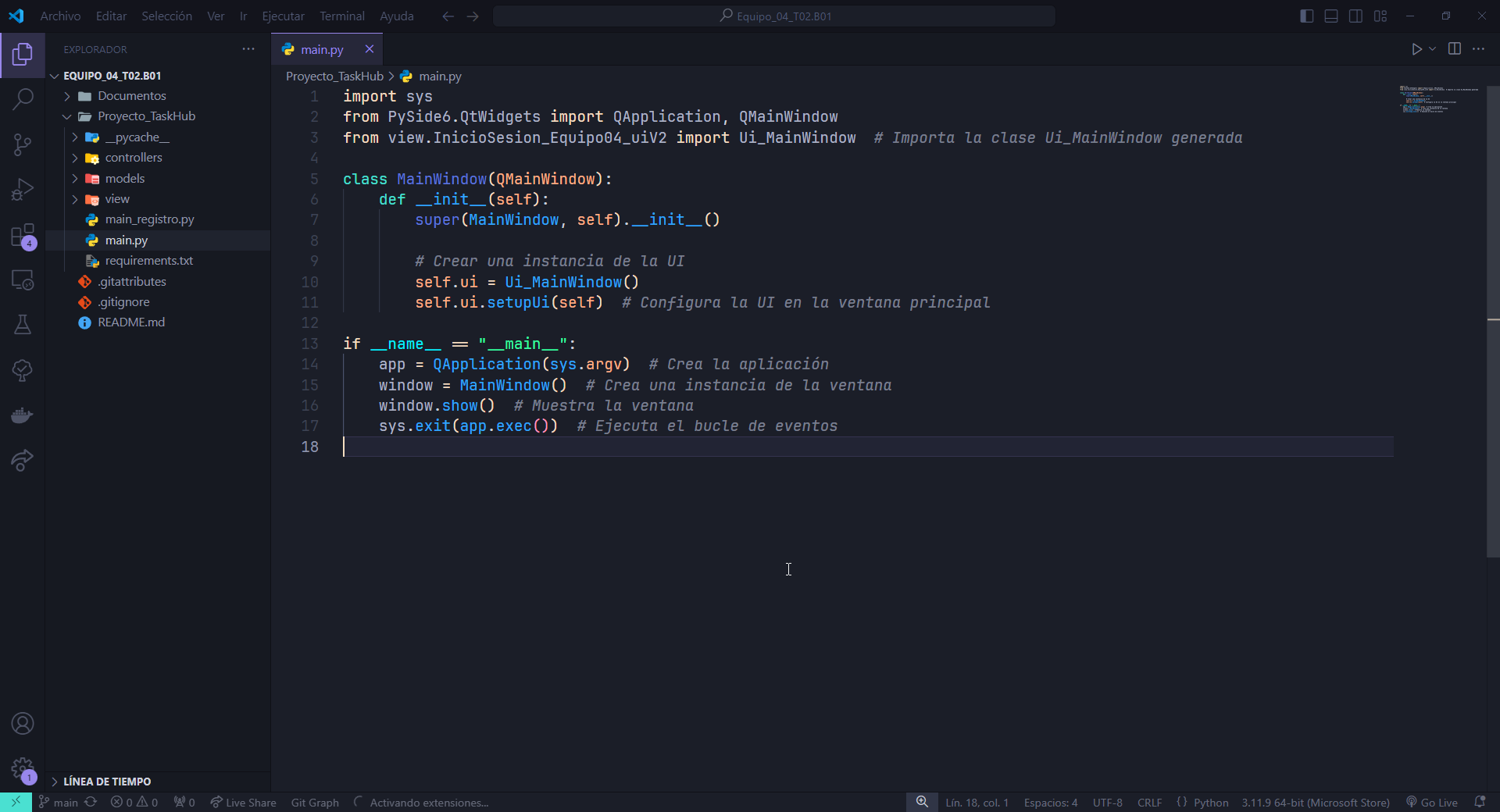
Para crear el entorno virtual, primero tenemos que tener un archivo requirements, este archivo contiene todas las dependencias que nos va a hacer falta en nuestro proyecto, en nuestro caso, nuestro archivo tiene las siguientes dependencias:



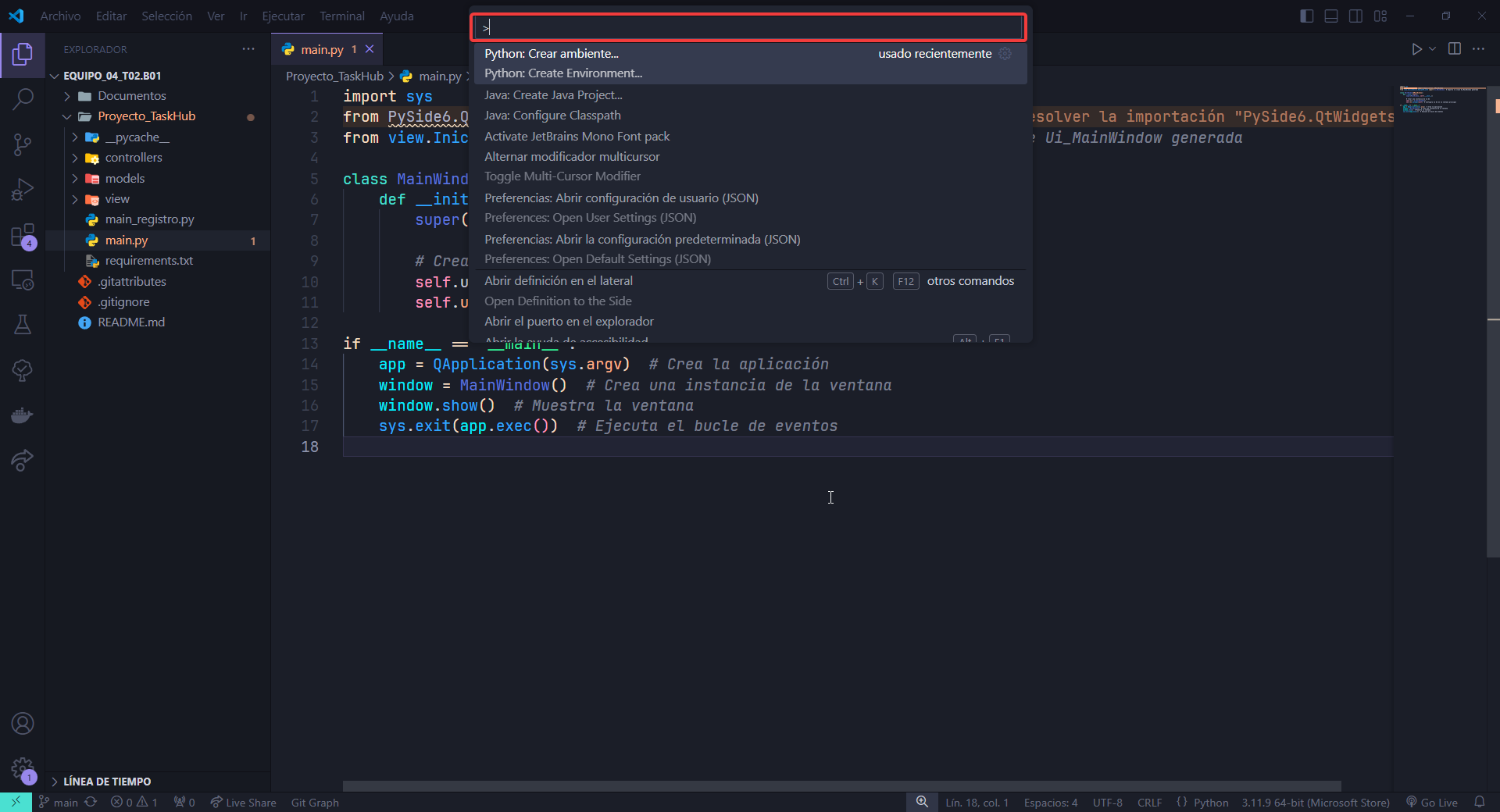
Una vez que tenemos nuestro archivo de dependencias localizado, podemos proceder a instalar nuestro entorno virtual.

Crear entorno de trabajo en Python:

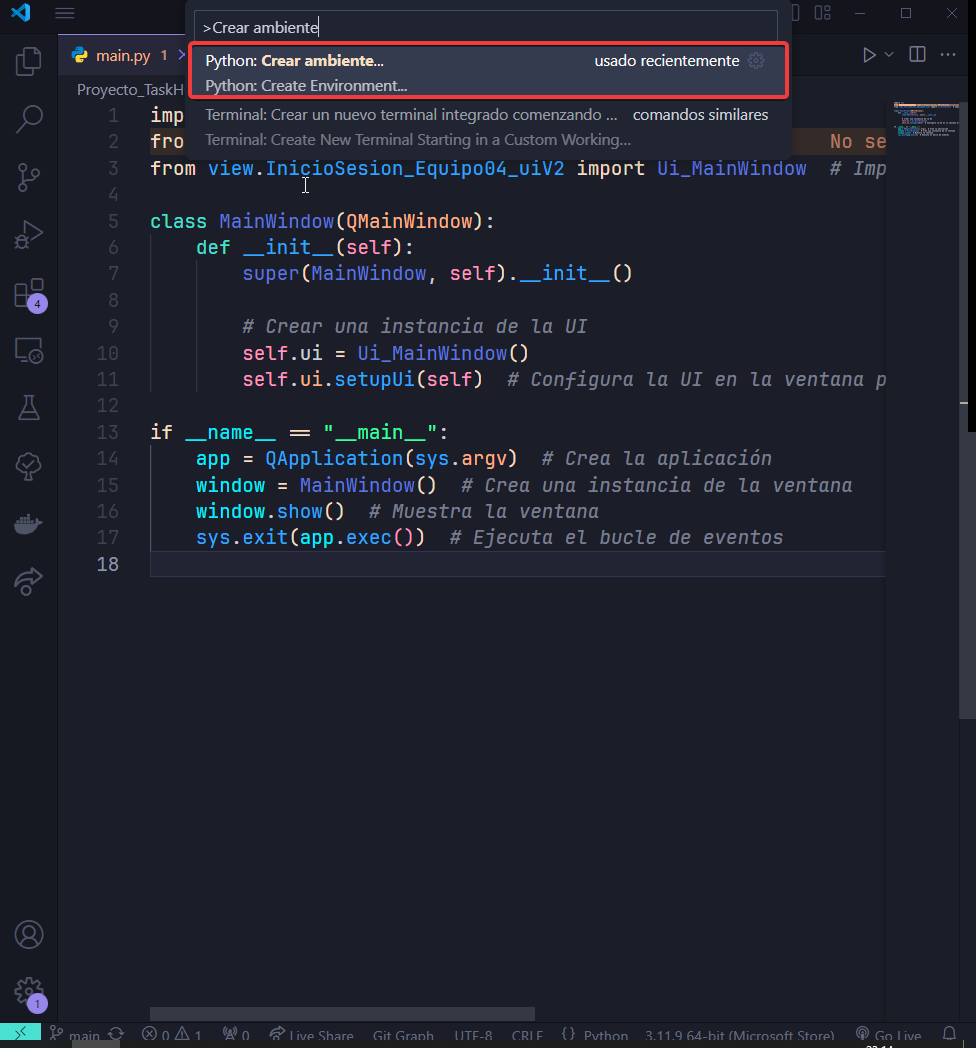
Lo primero que tenemos que hacer es ir a nuestro entorno de trabajo, en nuestro caso visual studio code:



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



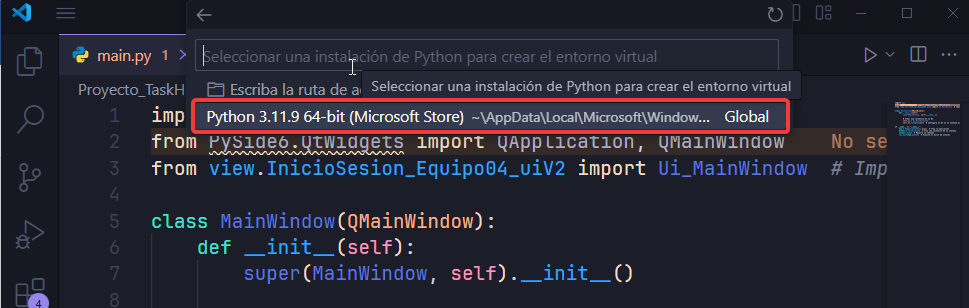
Una vez aquí buscaremos crear ambiente y pulsaremos sobre este:



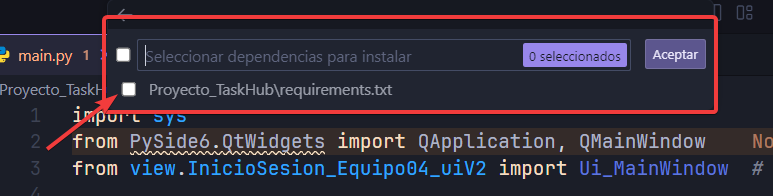
Ahora damos en Venv



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

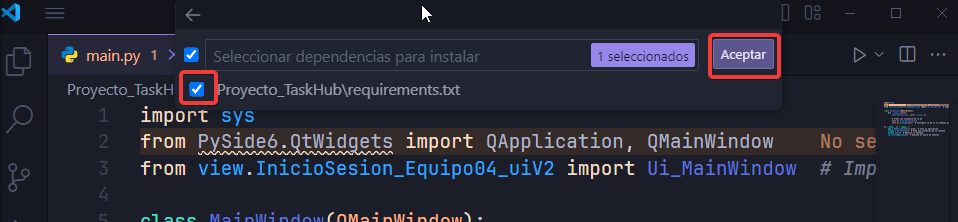


Ahora tendremos que marcar los requerimientos que nos hacen falta para nuestro proyecto, que aquí te instala todos los paquetes que necesites

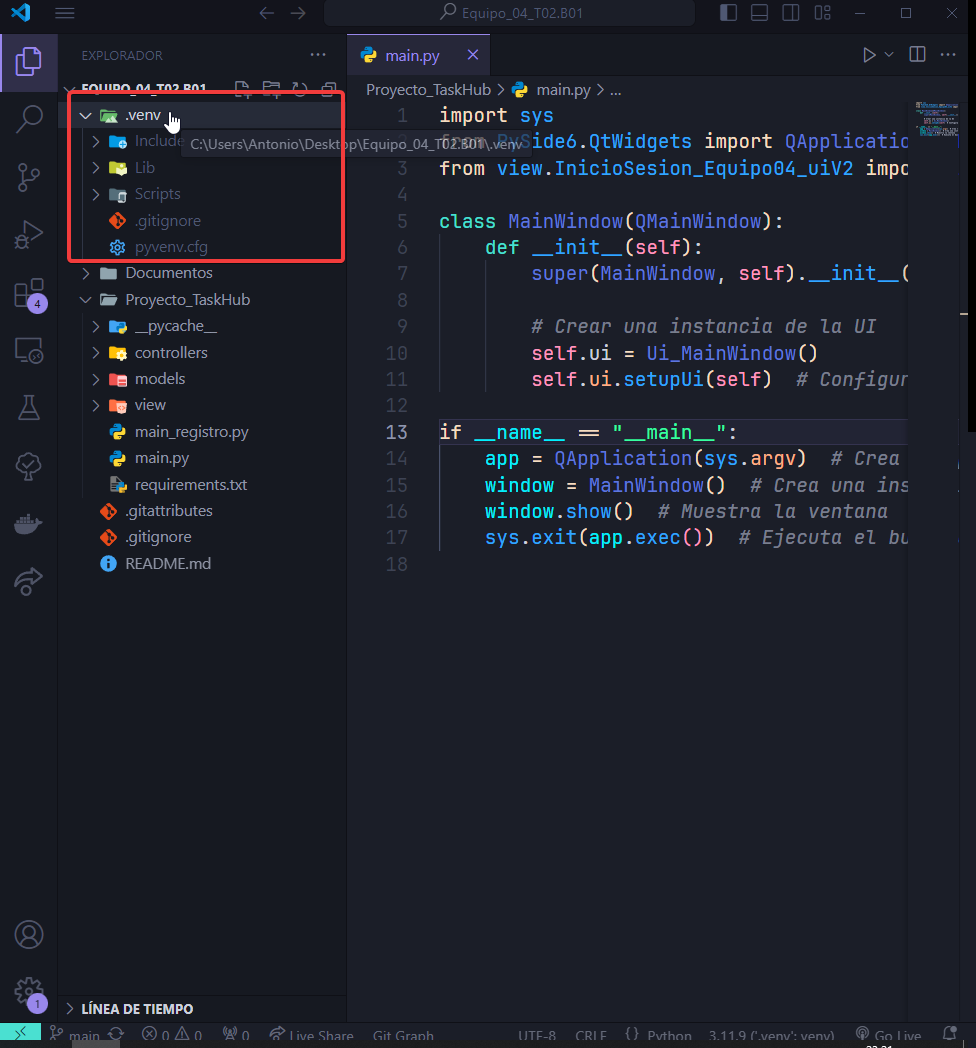


Este archivo es configurado por el usuario al cual le pertenece el proyecto, es muy útil ya que con esto no tenemos que pasar el entorno virtual ya que se descarga todos los requisitos

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.

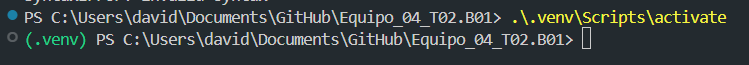
### Como hacerlo por comandos

python -m venv “nombre\_del\_entorno”

### Como activarlo:

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

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



### 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.

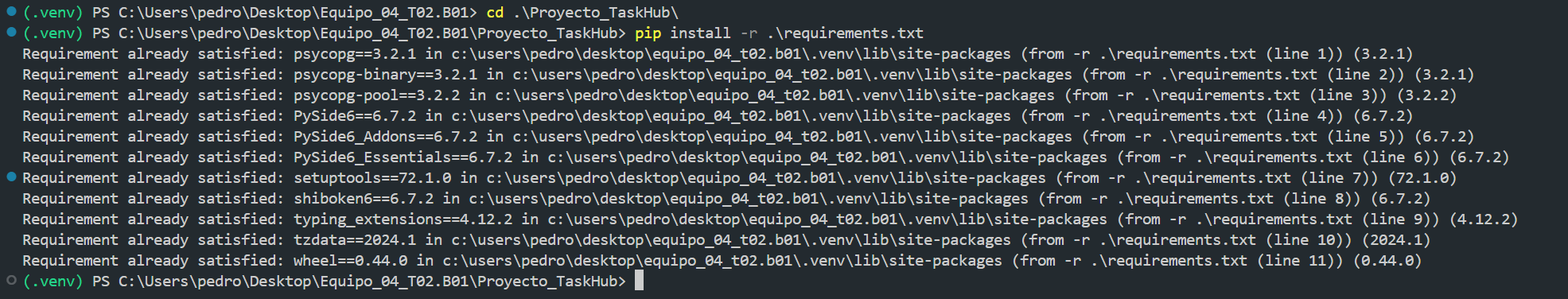
**typing\_extensions==4.12.2:** Proporciona compatibilidad con características de tipado avanzadas en Python, útiles en versiones de Python que no incluyen algunos de estos tipos de forma nativa.

**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

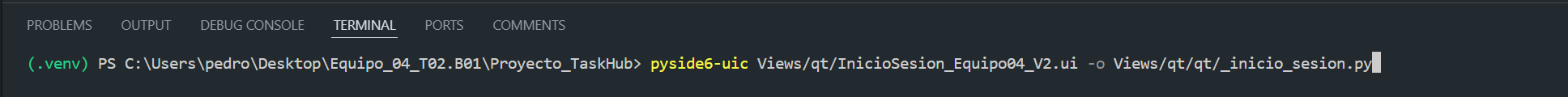
Para instalar las dependencias desde el archivo requirements.txt ejecutaremos el comando: pip install -r .\requirements.txt



## 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:



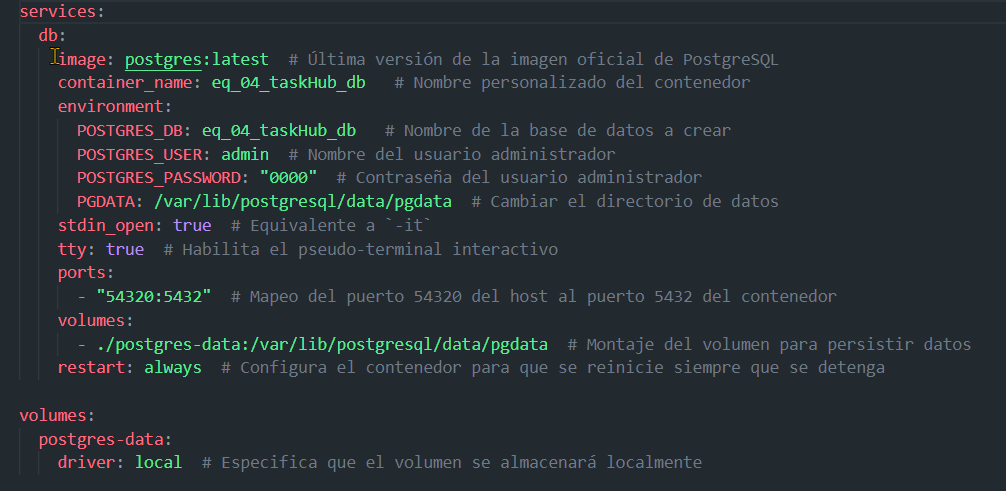
## DOCKER Y BBDD POSTGRESQL:

### 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:

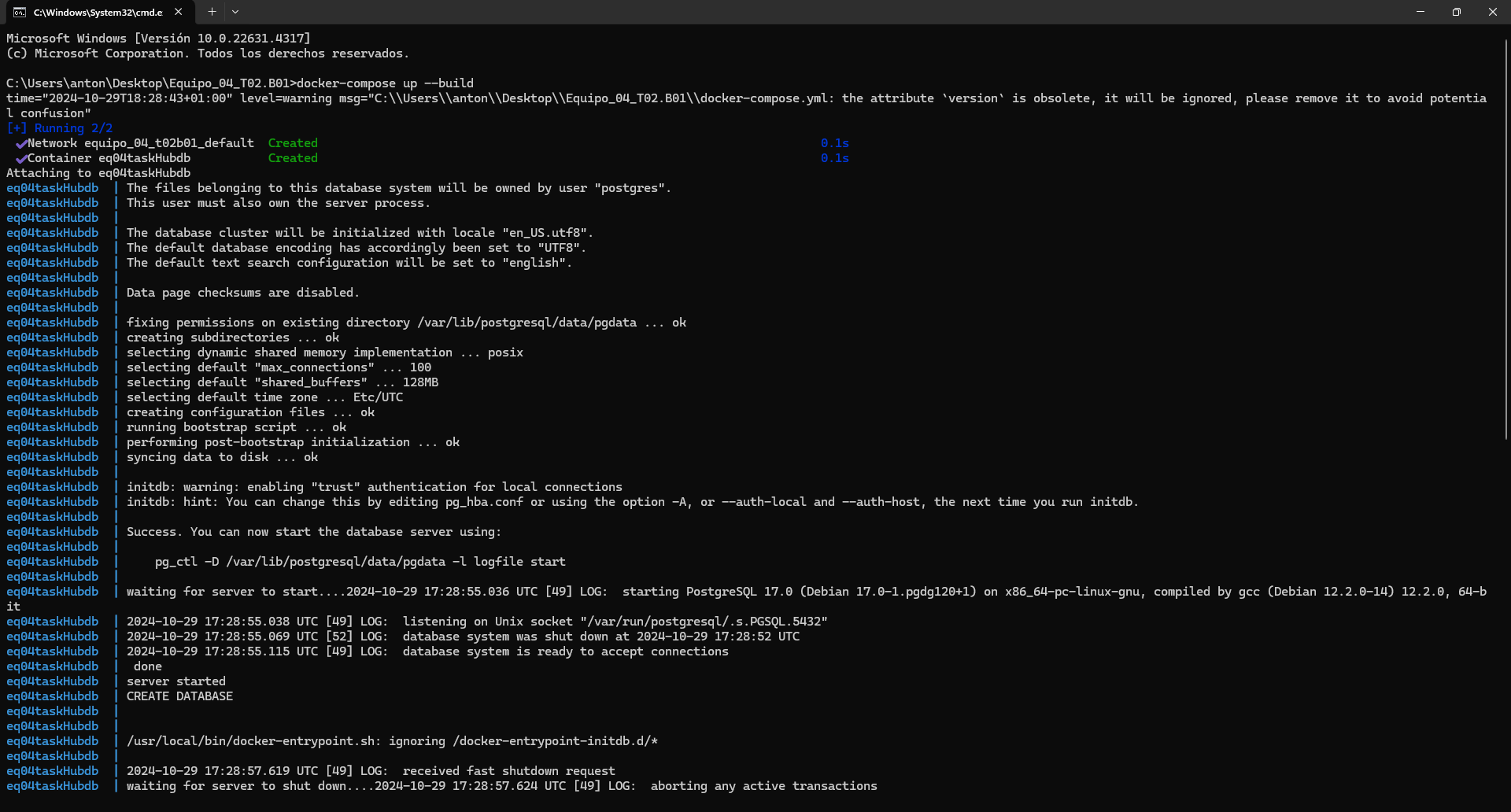
Aquí a destacar es que el nombre del contenedor se llama equipo eq\_04\_taskHub\_db y el nombre de la base de datos igual que el nombre del contenedor, el usuario es admin y la contraseña es 0000



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

Diagrama, Texto

Descripción generada automáticamente



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

Captura de pantalla de computadora

Descripción generada automáticamente

### 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.

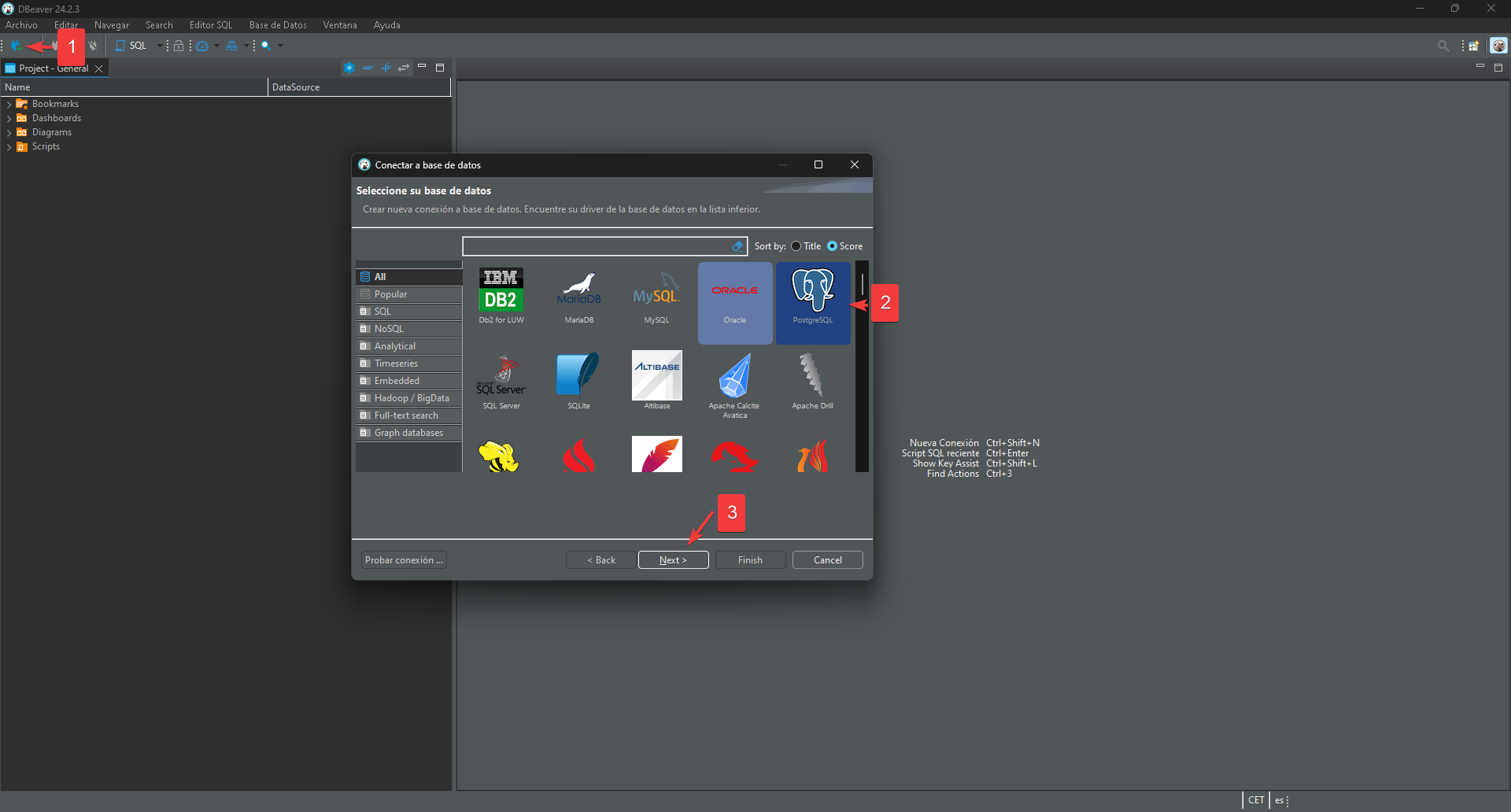
**restart: always:** Configura el contenedor para que se reinicie automáticamente en caso de fallos o si se reinicia el sistema.

**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:

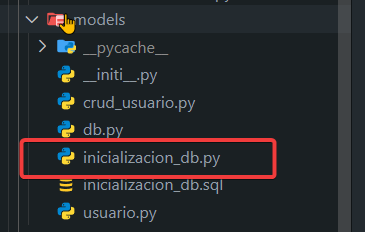
Interfaz de usuario gráfica

Descripción generada automáticamente

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.

