

**IES Carrillo Curso 24/25**

**Desarrollo de aplicaciones multiplataforma**

**Equipo 04:**

**Pedro González Martín**

**Antonio Gómez Camarena**

**David Castro Soriano**

**Tema 03 – Boletín 01**

indice

Contenido

[1. Configuración Inicial del Entorno 2](#_Toc183192148)

[1.1 Preparación del entorno virtual 2](#_Toc183192149)

[1.2 Configuración de Docker y la Base de Datos 2](#_Toc183192150)

[2. Creación del Modelo de Datos 4](#_Toc183192151)

[2.1 Estructura de la Base de Datos 4](#_Toc183192152)

[2.2 Conexión a la Base de Datos 4](#_Toc183192153)

[2.3 Inicialización de la Base de Datos 5](#_Toc183192154)

[2.4 Inserción de Datos 7](#_Toc183192155)

[2.5 Modelo de Datos en Python 10](#_Toc183192156)

[3. Diseño e Implementación de la Interfaz Gráfica. 11](#_Toc183192157)

[3.1 Creación Directa de la Ventana de Inicio de Sesión en Python 11](#_Toc183192158)

[3.1.1 Implementación del diseño 11](#_Toc183192159)

[3.2 Personalización y Mejora 20](#_Toc183192160)

[3.3 Ejecución de la Ventana 20](#_Toc183192161)

[4. Componentes Personalizados 21](#_Toc183192162)

[4.1 Empaquetado de componentes 25](#_Toc183192163)

[4.2 Instalar componentes personalizados 26](#_Toc183192164)

[5. Inclusión de componentes en interfaz 26](#_Toc183192165)

[6. Pruebas Unitarias 30](#_Toc183192166)

[7. Bibliografía 32](#_Toc183192167)

# 1. Configuración Inicial del Entorno

## 1.1 Preparación del entorno virtual

El primer paso es configurar un entorno virtual para aislar las dependencias del proyecto:

1. **Creación del entorno virtual**:

python -m venv venv

Esto crea un directorio llamado venv en la raíz del proyecto.

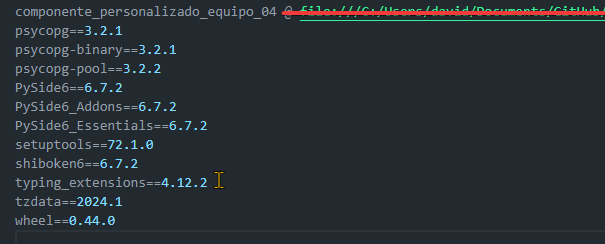
1. **Activación del entorno virtual**:

.\venv\Scripts\activate

1. **Instalación de dependencias**: El archivo requirements.txt contiene todas las dependencias del proyecto. Instálalas con:

pip install -r requirements.txt

Contenido de requirements.txt



Eliminamos esta línea porque al ser una línea absoluta, no funcionaría a otra persona.

1. **Verificación**: Comprueba que las dependencias están instaladas correctamente con:

pip list

## 1.2 Configuración de Docker y la Base de Datos

Usamos Docker para alojar la base de datos PostgreSQL. Esto asegura un entorno de base de datos consistente y fácil de configurar.

1. **Definir el archivo docker-compose.yml**: Este archivo describe el servicio PostgreSQL que se ejecutará dentro de un contenedor Docker:

version: '3.8'

services:

db:

image: postgres:latest

container\_name: eq\_04\_taskHub\_db

environment:

POSTGRES\_DB: eq\_04\_taskHub\_db

POSTGRES\_USER: admin

POSTGRES\_PASSWORD: "0000"

ports:

- "54320:5432"

volumes:

- ./postgres-data:/var/lib/postgresql/data

restart: no

1. **Iniciar el servicio**: Desde el directorio raíz del proyecto, ejecuta:

docker-compose up --build

Esto:

* + Descarga la imagen de PostgreSQL.
  + Crea un contenedor llamado eq\_04\_taskHub\_db.
  + Expone el puerto 5432 de la base de datos como 54320 en tu máquina local.

1. **Verificar la conexión**: Usa **DBeaver** para conectarte:
   * Host: localhost
   * Puerto: 54320
   * Usuario: admin
   * Contraseña: 0000
   * Base de datos: eq\_04\_taskHub\_db

# 2. Creación del Modelo de Datos

## **2.1 Estructura de la Base de Datos**

La base de datos incluye dos tablas principales: Usuarios y Tareas. Estas tablas están relacionadas mediante una clave foránea, donde cada tarea pertenece a un usuario.

**Definición de tablas (**inicializacion\_db.sql**)**:

CREATE TABLE IF NOT EXISTS usuarios (

            email VARCHAR(255) PRIMARY KEY,

            nombre\_usuario VARCHAR(255) NOT NULL,

            password VARCHAR(255) NOT NULL  )

 create table if not exists Tarea(

        nombre varchar(255) primary key,

        description varchar(255),

        idUsuario VARCHAR(255) not null,

        activa boolean default true,

        FOREIGN KEY(idUsuario) references Usuarios(email) on delete cascade

        )

## **2.2 Conexión a la Base de Datos**

El archivo db.py contiene funciones que centralizan la gestión de la conexión a la base de datos PostgreSQL, permitiendo que otras partes del proyecto interactúen con la base de datos de manera sencilla y reutilizable.

|  |
| --- |
| import psycopg  # Importamos el módulo `psycopg`, que es una biblioteca para trabajar con PostgreSQL desde Python.  # Función para conectar a la base de datos PostgreSQL  def connect\_db():      """      Conectar a la base de datos PostgreSQL y devolver la conexión.        :return: Objeto de conexión a la base de datos o None en caso de error.      """      try:          # Intentamos crear una conexión a la base de datos usando los datos proporcionados          conn = psycopg.connect(              dbname="eq\_04\_taskHub\_db",  # Nombre de la base de datos              user="admin",                 # Usuario de la base de datos              password="0000",              # Contraseña para acceder a la base de datos              host="localhost",             # Dirección del servidor de base de datos (local)              port="54320"                  # Puerto en el que se ejecuta PostgreSQL          )          return conn  # Si la conexión es exitosa, se devuelve el objeto `conn`      except Exception as error:          # Si ocurre algún error durante la conexión, se captura la excepción y se imprime un mensaje de error          print(f"Error al conectar con la base de datos: {error}")          return None  # En caso de error, se devuelve `None` para indicar que no se pudo conectar  # connect\_db  # Función para cerrar la conexión a la base de datos PostgreSQL  def close\_connection(conn):      """      Cerrar la conexión a la base de datos PostgreSQL.        :param conn: La conexión a la base de datos que se desea cerrar.      """      if conn is not None:  # Verificamos si la conexión existe (no es None)          try:              conn.close()  # Intentamos cerrar la conexión si está activa          except Exception as error:              # Si ocurre algún error al cerrar la conexión, se captura la excepción y se imprime un mensaje de error              print(f"Error al cerrar la conexión a la base de datos: {error}")  # close\_connection |

## **2.3 Inicialización de la Base de Datos**

El archivo inicializacion\_db.py contiene las funciones necesarias para preparar la base de datos del proyecto. Esto incluye la creación de tablas y la inserción de datos iniciales, asegurando que la estructura requerida para la aplicación esté lista antes de su uso.

|  |
| --- |
| 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          )          """,      )      commands2 = (          """          create table if not exists Tarea(          nombre varchar(255) primary key,          description varchar(255),          idUsuario VARCHAR(255) not null,          activa boolean default true,          FOREIGN KEY(idUsuario) references Usuarios(email) on delete cascade          )          """,      )      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 usuario creada.")              conn.close()  # Cerrar la conexión a la base de datos          else:              print("No se pudo conectar a la base de datos.")          conn2 = db.connect\_db()          if conn2 is not None:              with conn2.cursor() as cur2:                  for command2 in commands2:                      cur2.execute(command2)                  conn2.commit()                  print("Tabla de tarea creada correctamente")              conn2.close()          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 |

## **2.4 Inserción de Datos**

La función insert\_data en el archivo inicializacion\_db.py se encarga de insertar datos iniciales de ejemplo en la tabla Usuarios. Esto permite que la base de datos contenga registros básicos desde el inicio, útiles para pruebas o configuración inicial.def insert\_data():

|  |
| --- |
| # 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 = (          ('antonio@gmail.com', 'antonio', 'usuario0?'),          ('david@gmail.com', 'david', 'usuario0?'),          ('pedro@gmail.com', 'pedro', 'usuario0?')      )      insertsTables = (          ('Revisión de documentos', 'Revisar y corregir los documentos enviados por el cliente.', 'antonio@gmail.com',           "TRUE"),          ('Preparar presentación mensual', 'Crear presentación para la reunión mensual de resultados.',           'antonio@gmail.com', "TRUE"),          ('Actualizar base de datos', 'Actualizar la base de datos con los nuevos registros de clientes.',           'antonio@gmail.com', "FALSE"),          ('Reunión con el equipo', 'Coordinar reunión semanal con el equipo de desarrollo.', 'antonio@gmail.com', "TRUE"),          ('Enviar reporte de ventas', 'Elaborar y enviar el reporte de ventas mensual al gerente.', 'antonio@gmail.com',           "TRUE"),          ('Investigación de mercado', 'Analizar las tendencias del mercado para ajustar la estrategia.',           'antonio@gmail.com', "FALSE"),          ('Responder correos pendientes', 'Revisar y responder los correos electrónicos recibidos en la semana.',           'antonio@gmail.com', "TRUE"),          ('Revisión de inventario', 'Revisar el inventario de productos en el almacén.', 'david@gmail.com', "TRUE"),          ('Planificación de producción', 'Planificar las necesidades de producción para el próximo mes.',           'david@gmail.com', "TRUE"),          ('Capacitación del personal', 'Organizar una sesión de capacitación para el nuevo equipo.', 'david@gmail.com',           "FALSE"),          ('Análisis de costos', 'Analizar los costos de producción y buscar áreas de mejora.', 'david@gmail.com', "TRUE"),          ('Actualización de precios', 'Actualizar los precios de los productos según los nuevos costos.',           'david@gmail.com', "TRUE"),          (          'Contacto con proveedores', 'Revisar y confirmar las órdenes de compra con los proveedores.', 'david@gmail.com',          "TRUE"),          ('Elaboración de informe trimestral', 'Crear un informe con el desempeño del área en el trimestre.',           'david@gmail.com', "FALSE"),          ('Supervisión del proceso', 'Supervisar el proceso de ensamblaje en la planta.', 'david@gmail.com', "TRUE"),          ('Diseño de campaña publicitaria', 'Crear el diseño de la nueva campaña de marketing digital.',           'pedro@gmail.com', "TRUE"),          ('Revisión de redes sociales', 'Analizar el rendimiento de las publicaciones en redes sociales.',           'pedro@gmail.com', "FALSE"),          ('Creación de contenido', 'Desarrollar contenido para el blog de la empresa.', 'pedro@gmail.com', "TRUE"),          ('Revisión de SEO', 'Optimizar el SEO del sitio web de la empresa.', 'pedro@gmail.com', "TRUE"),          ('Coordinación con diseñadores', 'Reunirse con el equipo de diseño para revisar avances.', 'pedro@gmail.com',           "TRUE"),          (          'Análisis de métricas', 'Revisar las métricas de tráfico y conversión del sitio web.', 'pedro@gmail.com', "FALSE")      )        # 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.      insert\_query\_tablas = """          INSERT INTO Tarea (nombre, description, idUsuario, activa)          VALUES (%s, %s, %s, %s)          ON CONFLICT (nombre) DO NOTHING;      """      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.")          conn2 = db.connect\_db()          if conn2 is not None:              with conn2.cursor() as cur2:  # Crear un cursor para ejecutar comandos SQL                  for tablas in insertsTables:                      cur2.execute(insert\_query\_tablas, tablas)                  conn2.commit()                  print("Datos insertados correctamente.")              conn2.close()          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 |

## **2.5 Modelo de Datos en Python**

En el proyecto, cada tabla de la base de datos tiene una clase que la representa en Python. Estas clases encapsulan los atributos de las tablas y permiten trabajar con los datos de manera más estructurada en la aplicación. Clase Usuario (usuario.py):

La clase Usuario representa la tabla Usuarios en la base de datos. Cada instancia de esta clase corresponde a un registro en la tabla.

|  |
| --- |
| 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\_\_ |

La clase Tarea representa la tabla Tarea en la base de datos. Cada instancia de esta clase corresponde a un registro en la tabla.

|  |
| --- |
| class Tarea:      """      Clase que representa una tarea.      """      def \_\_init\_\_(self, nombre, descripcion, idusuario, activa):          self.\_nombre = nombre          self.\_descripcion = descripcion          self.\_idusuario = idusuario          self.\_activa = activa |

# 3. Diseño e Implementación de la Interfaz Gráfica.

La interfaz gráfica de usuario (GUI) está implementada con PySide6, utilizando el patrón MVC. A continuación, describo cómo se diseñaron las ventanas y componentes, cómo se integraron en el proyecto y cómo interactúan con la lógica de negocio.

## **3.1 Creación Directa de la Ventana de Inicio de Sesión en Python**

En lugar de usar **Qt Designer** para generar el diseño, la ventana de inicio de sesión se creó manualmente en Python utilizando las clases de **PySide6**. Esto permite un mayor control sobre la personalización y el comportamiento dinámico de los elementos de la interfaz.

### 3.1.1 Implementación del diseño

La ventana se creó mediante una subclase de QMainWindow, donde se definen y configuran los widgets como campos de texto y botones.

**Ventana de Inicio de Sesión (**login\_window.py**):** Este archivo integra la interfaz gráfica generada con la lógica de negocio:

|  |
| --- |
| from PySide6.QtWidgets import QMainWindow, QMessageBox  # Importar QApplication para la app y QMainWindow para la ventana principal  from PySide6.QtCore import Slot  # Importar Slot para los decoradores de los métodos  from views.qt.qt\_inicio\_sesion import Ui\_Inicio\_Sesion\_Equipo04 # Importar la clase generada a partir del archivo .ui  from views.registro\_window import RegistroWindow  # Importar la clase de la ventana de registro  from controllers.usuario\_controller import UsuarioController  # Importar el controlador del usuario  from views.view\_tareas\_windows import View\_Tarea\_Windows  from utils import variables  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}")              variables.usuario = name              # pasamos a la siguiente ventana              self.view\_tarea = View\_Tarea\_Windows()              self.hide()              self.view\_tarea.show()          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 pontalla en los campos de registros      """      @Slot()      def vaciarCamposDeTexto(self):          print("Borrar textos")          self.ui.texto\_usuario\_correo.setText("")          self.ui.texto\_contrasenna.setText("")      # vaciarCamposDeTexto |

**Ventana de Registro (**registro\_window.py**):** Similar a la ventana de inicio de sesión, permite a los usuarios registrarse.

|  |
| --- |
| from PySide6.QtWidgets import QMainWindow, QMessageBox, QApplication  # Importar la clase QMainWindow para crear la ventana principal  from PySide6.QtCore import Slot  # Importar Slot para la conexión de señales y slots  from views.qt.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 del usuario  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)          self.ui.btn\_registro.clicked.connect(self.function\_registro)          self.ui.vaciar\_campos\_de\_texto.triggered.connect(self.vaciarCamposDeTexto)          self.ui.action\_nuestra\_empresa\_2.triggered.connect(self.abrirAcercaDe)            self.usuario\_controller = UsuarioController()        # \_\_init\_\_        @Slot()      def function\_volver\_iniciar\_sesion(self):          self.hide()            if self.parent() is not None:              self.parent().mostrar\_login()        @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")              mensaje\_error = QMessageBox(self)              mensaje\_error.setWindowTitle("Error las contraseñas no coinciden")              mensaje\_error.setText("Error en el registro, las contraseñas no coinciden")              mensaje\_error.setIcon(QMessageBox.Critical)              mensaje\_error.exec()              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()          """      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 pontalla en los capos de registros      """      @Slot()      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        def closeEvent(self, event):          """Cerrar la aplicación completa al cerrar la ventana de registro."""          QApplication.quit() |

**Ventana de Tareas (**view\_tareas\_windows.py**):** Permite listar, buscar y gestionar tareas mediante una tabla y barra de búsqueda.

|  |
| --- |
| from PySide6.QtWidgets import QWidget, QHBoxLayout, QVBoxLayout, QTableWidget, QTableWidgetItem, QHeaderView, QCheckBox  from Componentes\_Personalizado import Search\_Bar, Button\_Search  from PySide6.QtCore import Slot  from utils import variables  from controllers.tarea\_controller import TareaController  from models.Tarea import Tarea  from PySide6.QtGui import Qt  class View\_Tarea\_Windows(QWidget):      def \_\_init\_\_(self, parent=None):          super().\_\_init\_\_(parent)          # Layout principal          self.resize(500, 500)          self.layout\_vertical\_main = QVBoxLayout()          self.setLayout(self.layout\_vertical\_main)  # Establecemos el layout por defecto del widget          # Layout horizontal para barra de búsqueda          self.layout\_horizontal\_search\_bar = QHBoxLayout()          self.button\_search = Button\_Search(icon=variables.iconSearch, text="Buscar")          self.search\_bar = Search\_Bar()          # Creamos la tabla          self.tabla\_tarea = QTableWidget()          self.tabla\_tarea.setColumnCount(3)          self.tabla\_tarea.setHorizontalHeaderLabels(["Nombre", "Descripción", "Activa"])          self.header = self.tabla\_tarea.horizontalHeader()          self.header.setSectionResizeMode(QHeaderView.Stretch)          # Añadir componentes al layout horizontal          self.layout\_horizontal\_search\_bar.addWidget(self.search\_bar)          self.layout\_horizontal\_search\_bar.addWidget(self.button\_search)          # Añadir layout horizontal al layout principal          self.layout\_vertical\_main.addLayout(self.layout\_horizontal\_search\_bar)          self.layout\_vertical\_main.addWidget(self.tabla\_tarea)          # Variables          self.tareas\_originales = []  # Para almacenar las tareas originales sin filtrar          # Obtener datos iniciales          self.obtenerDatos()          # Conectar eventos          self.search\_bar.textEdited.connect(self.cambioEnTexto)          self.button\_search.signal\_presionado.connect(self.cambioEnTexto)      def llenar\_tabla(self, datos):          """          Llenar la tabla con los datos proporcionados.          """          self.tabla\_tarea.setRowCount(len(datos))  # Establecer el número de filas          for fila, datos\_fila in enumerate(datos):              for columna, dato in enumerate(datos\_fila):                  if columna == 2:  # Columna "Activa"                      checkbox = QCheckBox()                      checkbox.setChecked(dato == "No")  # Marcar si la tarea no está activa                      checkbox.setStyleSheet("QCheckBox::indicator { subcontrol-position: center; }")                      checkbox.setEnabled(False)                      self.tabla\_tarea.setCellWidget(fila, columna, checkbox)                  else:                      item = QTableWidgetItem(dato)  # Crear un QTableWidgetItem                      #item.setTextAlignment(Qt.AlignCenter)  # Centrar el texto                      self.tabla\_tarea.setItem(fila, columna, item)  # Añadir el item a la celda correspondiente                  # Si la tarea no está activa, aplicar estilos                  if columna == 2 and dato == "No":                      for c in range(3):  # Aplicar estilos a todas las celdas de la fila                          item = self.tabla\_tarea.item(fila, c)                          if item:                              item.setForeground(Qt.red)  # Color rojo                              font = item.font()                              font.setStrikeOut(True)  # Texto tachado                              item.setFont(font)        @Slot()      def obtenerDatos(self):          """          Obtener las tareas del usuario y llenar la tabla.          """          try:              # Llamar al controlador para obtener las tareas del usuario              tareas = TareaController().obtener\_tareas\_por\_usuario(variables.usuario)              # Guardar las tareas originales para poder filtrar              self.tareas\_originales = tareas              # Transformar las tareas en un formato adecuado para la tabla              datos = [                  [tarea.nombre, tarea.descripcion, "Sí" if tarea.activa else "No"]                  for tarea in tareas              ]              # Llenar la tabla con los datos obtenidos              self.llenar\_tabla(datos)          except Exception as e:              print(f"Error al obtener las tareas: {e}")      @Slot()      def cambioEnTexto(self):          """          Actualizar la tabla en base al texto ingresado en la barra de búsqueda.          """          print("Se esta cambiando los datos (Boton o searchBar)")          texto = self.search\_bar.text().lower()  # Convertir a minúsculas para comparación sin distinción de mayúsculas          tareas\_filtradas = [              tarea              for tarea in self.tareas\_originales              if texto in tarea.nombre.lower()  # Filtrar por nombre de tarea          ]          # Transformar las tareas filtradas en un formato adecuado para la tabla          datos = [              [tarea.nombre, tarea.descripcion, "Sí" if tarea.activa else "No"]              for tarea in tareas\_filtradas          ]          # Llenar la tabla con los datos filtrados          self.llenar\_tabla(datos) |

## **3.2 Personalización y Mejora**

**Estilos Visuales:** Se pueden añadir estilos personalizados usando hojas de estilo de Qt:

self.setStyleSheet("""

QLineEdit {

border: 1px solid #ccc;

border-radius: 5px;

padding: 5px;

}

QPushButton {

background-color: #4CAF50;

color: white;

border: none;

border-radius: 5px;

padding: 10px;

}

QPushButton:hover {

background-color: #45a049;

}

""")

## **3.3 Ejecución de la Ventana**

Para probar esta ventana, simplemente puedes instanciarla desde el archivo main.py:

|  |
| --- |
| 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 méto\_do 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\_\_ |

# **4. Componentes Personalizados**

**Botón de Búsqueda (**button\_search.py**):**

El componente Button\_Search extiende la funcionalidad del widget base QPushButton, es un botón personalizado tanto en apariencia como en comportamiento. Permite especificar características como texto, familia tipográfica, tamaño de fuente, colores (de texto, fondo y presionado), radio de borde e íconos, lo que lo convierte en un elemento adaptable a múltiples necesidades dentro de la interfaz gráfica. Además, el componente emite una señal personalizada signal\_presionado, facilitando la comunicación entre el botón y otros elementos de la aplicación.

En el constructor, se configuran los valores iniciales del botón, incluyendo las propiedades visuales como el color de fondo (bg\_color) y la fuente mediante los métodos updateBackgroundColor y setFontProperties. Si se proporciona un ícono, el botón verifica que la ruta del archivo sea válida antes de cargarlo con QIcon y establecer su tamaño mediante setIconSize. Esto asegura que no se generen errores al intentar cargar íconos inexistentes, mostrando un mensaje de advertencia si la ruta es incorrecta.

La funcionalidad principal se centra en la interacción visual y los eventos personalizados. El método mousePressEvent sobrescribe el comportamiento base del botón para emitir la señal signal\_presionado cuando se detecta un clic, manteniendo la funcionalidad predeterminada del evento original mediante una llamada al método de la clase base. Esto permite que el botón informe a otros elementos de la interfaz sobre la interacción del usuario, sin necesidad de modificar el evento clicked base.

En cuanto al diseño, el botón ofrece herramientas para actualizar dinámicamente sus propiedades visuales. Por ejemplo, setFontProperties permite cambiar la familia y el tamaño de la fuente, además de aplicar negrita, mientras que updateBackgroundColor modifica el color de fondo del botón a través de su paleta. Este enfoque modular facilita su integración en diferentes partes de la aplicación, manteniendo coherencia visual y flexibilidad funcional en un único componente reutilizable.

|  |
| --- |
| import os  from PySide6.QtWidgets import QPushButton  from PySide6.QtGui import QPalette, QColor, QIcon  from PySide6.QtCore import Signal, QSize  from .utils import utils\_colores as colores  class Button\_Search(QPushButton):      signal\_presionado = Signal()      def \_\_init\_\_(          self,          text="Buscar",          font\_family="Calibri",          font\_size=20,          font\_color=colores.BLACK,          bg\_color=colores.CORAL,          pressed\_color=colores.YELLOW\_ORANGE,          border\_radius=5,          icon="search.png",          parent=None,      ):          super().\_\_init\_\_(text,parent)          self.bg\_color = QColor(bg\_color)          self.setFontProperties(font\_family, font\_size)          self.updateBackgroundColor(self.bg\_color)          # Verifica si la ruta del icono es correcta          icon\_path = icon          if not os.path.exists(icon\_path):              print(f"Error: No se encontró el archivo en la ruta {icon\_path}")          else:              # Cargar el icono SVG directamente usando QIcon              icono = QIcon(icon\_path)              self.setIcon(icono)              self.setIconSize(QSize(24, 24))  # Ajusta el tamaño del icono según sea necesario      def setFontProperties(self, family, size):          """          Configura la fuente del botón.          :param family: Nombre de la familia tipográfica.          :param size: Tamaño de la fuente en puntos.          """          font = self.font()  # Obtiene la fuente actual del botón          font.setBold(True) # Establecemos la fuente en negrita          font.setFamily(family)  # Establece la familia de la fuente          font.setPointSize(size)  # Establece el tamaño de la fuente          self.setFont(font)  # Aplica la configuración de la fuente al botón      # Método para actualizar el color de fondo del botón      def updateBackgroundColor(self, color):          """          Actualiza el color de fondo del botón.          :param color: Objeto QColor que representa el color de fondo.          """          palette = self.palette()  # Obtiene la paleta actual del botón          palette.setColor(QPalette.Button, color)  # Configura el color de fondo del botón en la paleta          self.setPalette(palette)  # Aplica la paleta actualizada al botón          self.setAutoFillBackground(True)  # Permite que el botón rellene automáticamente su fondo        #############      ## EVENTOS ##      #############      """      Eventos personalizados que modifican el comportamiento visual del botón y emiten señales cuando se produce interacción.      """      # Evento que detecta cuando el botón es presionado      def mousePressEvent(self, event):          """          Cambia el color de fondo al pressed\_color y emite la señal signal\_presionado cuando el botón es presionado.          :param event: Evento de presionar el botón.          """          self.signal\_presionado.emit()  # Emite la señal personalizada de botón presionado          super().mousePressEvent(event)  # Llama al método original de mousePressEvent de la clase base |

Nuestro componente personalizado se vería así:



**Barra de Búsqueda (**search\_bar.py**):**

La clase Search\_Bar es un componente personalizado que extiende QLineEdit para proporcionar una barra de búsqueda con un diseño y configuración de fuente específicos. Este componente está pensado para facilitar la entrada de texto en la interfaz gráfica.

En el constructor, se configura la fuente de la barra de búsqueda utilizando setFont, donde se especifica la familia tipográfica como "Calibri" y un tamaño de fuente de 20 puntos. Esto asegura que el texto introducido sea claro y consistente en términos de diseño. Además, se aplica un estilo visual mediante setStyleSheet, añadiendo un borde sólido en color naranja (#f2784b), junto con un borde redondeado con un radio de 5 píxeles.

El diseño de Search\_Bar permite que este componente sea reutilizado en diferentes partes del proyecto.

|  |
| --- |
| from PySide6.QtWidgets import QLineEdit  class Search\_Bar(QLineEdit):      def \_\_init\_\_(self):          super().\_\_init\_\_()          self.leText = QLineEdit()          # Ajustamos la fuente:          font = self.font()          font.setFamily("Calibri")          font.setPointSize(20)          self.setFont(font)          self.setStyleSheet("border: 2px solid #f2784b;"                                    "border-radius:5px;") |

Nuestra barra de búsqueda quedaría así:



## 4.1 Empaquetado de componentes

Para poder crear el empaquetado necesitamos configurarlo a través de un archivo setup.py, este archivo contiene la configuración del empaquetado.

|  |
| --- |
| from setuptools import setup, find\_packages  setup(      name="componente\_personalizado\_equipo\_04",      version="0.1",      description="Coponentes personalizados con PySide6",      author="Equipo\_04",      author\_email="Equipo\_04@equipo04.com",      packages=find\_packages(),      install\_requires=[          "PySide6"      ],      python\_requires=">=3.6",      include\_package\_data=True,  # Esto permite incluir archivos adicionales  ) |

También necesitamos un archivo MANIFEST.in en el que vamos a incluir los archivos que nos hacen falta para que funcionen nuestros componentes.

|  |
| --- |
| include Componentes\_Personalizado/utils/icons/search-512.png  include Componentes\_Personalizado/utils/utils\_colores.py |

En este caso a necesitamos estas propiedades.

El archivo \_\_init\_\_.py tiene la importación de los componentes que vamos a empaquetar.

|  |
| --- |
| from Componentes\_Personalizado.custom\_qposhbutton import Button\_Search  from Componentes\_Personalizado.Search\_Bar import Search\_Bar |

Para empaquetar los componentes, lo haremos con el siguiente comando:

python .\setup.py sdist

## 4.2 Instalar componentes personalizados

**Para instalar los nuevos componentes utilizaremos el siguiente comando:** *pip install .\Proyecto\_TaskHub\components\componente\_personalizado\_equipo\_04-0.1.tar.gz*

# **5. Inclusión de componentes en interfaz**

**En la clase View\_Tarea\_Windows, hemos creado una ventana que incluye los nuevos componentes personalizados Search\_Bar y Button\_Search, diseñados para mejorar la experiencia de usuario y centralizar la funcionalidad de búsqueda de tareas. Para integrar estos componentes en nuestra interfaz, los hemos empaquetado en la librería “Componentes\_Personalizado” y los hemos importado directamente al módulo. Esto nos permite mantener un diseño modular y reutilizable en el desarrollo de la aplicación.**

**En el constructor de la clase, ambos componentes se instancian y personalizan. A Button\_Search se le asigna un ícono y texto, mientras que Search\_Bar se configura para capturar la entrada de texto. Ambos se organizan dentro de un layout horizontal (QHBoxLayout), que luego se agrega al layout principal vertical (QVBoxLayout).**

**Para conectar los eventos de búsqueda, se establecieron señales: signal\_presionado del botón y textEdited de la barra de búsqueda. Ambos eventos disparan el método cambioEnTexto, lo que permite actualizar dinámicamente la tabla según el texto ingresado o al presionar el botón. Este diseño asegura que la funcionalidad de búsqueda sea accesible tanto desde la interacción textual como desde un clic.**

|  |
| --- |
| from PySide6.QtWidgets import QWidget, QHBoxLayout, QVBoxLayout, QTableWidget, QTableWidgetItem, QHeaderView, QCheckBox  from Componentes\_Personalizado import Search\_Bar, Button\_Search  from PySide6.QtCore import Slot  from utils import variables  from controllers.tarea\_controller import TareaController  from models.Tarea import Tarea  from PySide6.QtGui import Qt  class View\_Tarea\_Windows(QWidget):      def \_\_init\_\_(self, parent=None):          super().\_\_init\_\_(parent)          # Layout principal          self.resize(500, 500)          self.layout\_vertical\_main = QVBoxLayout()          self.setLayout(self.layout\_vertical\_main)  # Establecemos el layout por defecto del widget          # Layout horizontal para barra de búsqueda          self.layout\_horizontal\_search\_bar = QHBoxLayout()          self.button\_search = Button\_Search(icon=variables.iconSearch, text="Buscar")          self.search\_bar = Search\_Bar()          # Creamos la tabla          self.tabla\_tarea = QTableWidget()          self.tabla\_tarea.setColumnCount(3)          self.tabla\_tarea.setHorizontalHeaderLabels(["Nombre", "Descripción", "Activa"])          self.header = self.tabla\_tarea.horizontalHeader()          self.header.setSectionResizeMode(QHeaderView.Stretch)          # Añadir componentes al layout horizontal          self.layout\_horizontal\_search\_bar.addWidget(self.search\_bar)          self.layout\_horizontal\_search\_bar.addWidget(self.button\_search)          # Añadir layout horizontal al layout principal          self.layout\_vertical\_main.addLayout(self.layout\_horizontal\_search\_bar)          self.layout\_vertical\_main.addWidget(self.tabla\_tarea)          # Variables          self.tareas\_originales = []  # Para almacenar las tareas originales sin filtrar          # Obtener datos iniciales          self.obtenerDatos()          # Conectar eventos          self.search\_bar.textEdited.connect(self.cambioEnTexto)          self.button\_search.signal\_presionado.connect(self.cambioEnTexto)      def llenar\_tabla(self, datos):          """          Llenar la tabla con los datos proporcionados.          """          self.tabla\_tarea.setRowCount(len(datos))  # Establecer el número de filas          for fila, datos\_fila in enumerate(datos):              for columna, dato in enumerate(datos\_fila):                  if columna == 2:  # Columna "Activa"                      checkbox = QCheckBox()                      checkbox.setChecked(dato == "No")  # Marcar si la tarea no está activa                      checkbox.setStyleSheet("QCheckBox::indicator { subcontrol-position: center; }")                      checkbox.setEnabled(False)                      self.tabla\_tarea.setCellWidget(fila, columna, checkbox)                  else:                      item = QTableWidgetItem(dato)  # Crear un QTableWidgetItem                      #item.setTextAlignment(Qt.AlignCenter)  # Centrar el texto                      self.tabla\_tarea.setItem(fila, columna, item)  # Añadir el item a la celda correspondiente                  # Si la tarea no está activa, aplicar estilos                  if columna == 2 and dato == "No":                      for c in range(3):  # Aplicar estilos a todas las celdas de la fila                          item = self.tabla\_tarea.item(fila, c)                          if item:                              item.setForeground(Qt.red)  # Color rojo                              font = item.font()                              font.setStrikeOut(True)  # Texto tachado                              item.setFont(font)        @Slot()      def obtenerDatos(self):          """          Obtener las tareas del usuario y llenar la tabla.          """          try:              # Llamar al controlador para obtener las tareas del usuario              tareas = TareaController().obtener\_tareas\_por\_usuario(variables.usuario)              # Guardar las tareas originales para poder filtrar              self.tareas\_originales = tareas              # Transformar las tareas en un formato adecuado para la tabla              datos = [                  [tarea.nombre, tarea.descripcion, "Sí" if tarea.activa else "No"]                  for tarea in tareas              ]              # Llenar la tabla con los datos obtenidos              self.llenar\_tabla(datos)          except Exception as e:              print(f"Error al obtener las tareas: {e}")      @Slot()      def cambioEnTexto(self):          """          Actualizar la tabla en base al texto ingresado en la barra de búsqueda.          """          print("Se esta cambiando los datos (Boton o searchBar)")          texto = self.search\_bar.text().lower()  # Convertir a minúsculas para comparación sin distinción de mayúsculas          tareas\_filtradas = [              tarea              for tarea in self.tareas\_originales              if texto in tarea.nombre.lower()  # Filtrar por nombre de tarea          ]          # Transformar las tareas filtradas en un formato adecuado para la tabla          datos = [              [tarea.nombre, tarea.descripcion, "Sí" if tarea.activa else "No"]              for tarea in tareas\_filtradas          ]          # Llenar la tabla con los datos filtrados          self.llenar\_tabla(datos) |

# **6. Pruebas Unitarias**

Es fundamental realizar pruebas para garantizar el correcto funcionamiento del proyecto. Usamos pytest y pytest-qt para realizar pruebas unitarias y de integración.

Para realizar las pruebas necesitamos instalar la siguiente librería:

pip install pytest pytest-qt

Y usaremos el siguiente comando:

pytest -v -s .\Proyecto\_TaskHub\unit\_test\test\_component.py

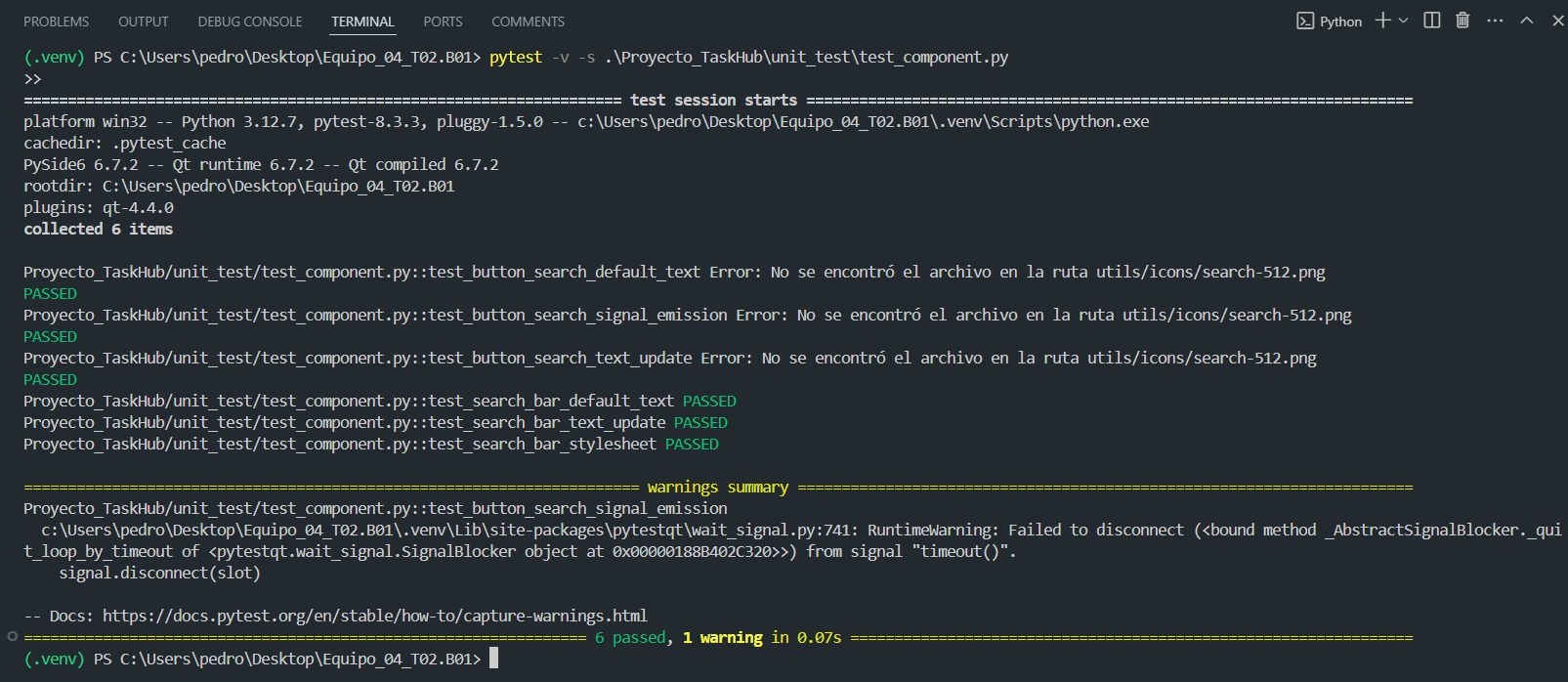
Las pruebas realizadas a los componentes Button\_Search y Search\_Bar se centran en validar su comportamiento esperado y garantizar que cumplan con sus objetivos funcionales y visuales. Estas pruebas utilizan pytest junto con qtbot para simular interacciones del usuario y verificar que los resultados sean correctos.

Para Button\_Search, se han definido tres pruebas clave. La primera prueba, test\_button\_search\_default\_text, verifica que el texto predeterminado del botón sea "Buscar". Esto asegura que el componente se inicializa correctamente con el texto esperado. La segunda prueba, test\_button\_search\_signal\_emission, evalúa si la señal personalizada signal\_presionado se emite correctamente al hacer clic en el botón, simulando un clic con qtbot.mouseClick. Por último, test\_button\_search\_text\_update comprueba que el texto del botón puede actualizarse dinámicamente, lo que es fundamental para su reutilización en diferentes contextos.

En el caso de Search\_Bar, se han realizado otras tres pruebas. La primera, test\_search\_bar\_default\_text, valida que el texto inicial de la barra de búsqueda esté vacío, lo cual es importante para garantizar que no haya información previa al momento de usarla. La segunda prueba, test\_search\_bar\_text\_update, simula la escritura de texto en la barra utilizando qtbot.keyClicks y verifica que el texto ingresado se actualice correctamente. Esto confirma que la barra responde adecuadamente a las interacciones del usuario. Finalmente, test\_search\_bar\_stylesheet valida que el estilo visual aplicado al componente, definido mediante setStyleSheet, coincida con el esperado. Esto asegura que el diseño del componente sea consistente con las especificaciones.

Estas pruebas cubren tanto los aspectos funcionales como visuales de los componentes, garantizando su correcto funcionamiento en la interfaz y su capacidad para reaccionar de manera confiable a las acciones del usuario.

|  |
| --- |
| import pytest  from PySide6.QtWidgets import QApplication  from Componentes\_Personalizado import Button\_Search, Search\_Bar  from PySide6.QtCore import Qt  @pytest.fixture(scope="session")  # La aplicación existirá durante toda la sesión de pytest  def app():      """Fixture para inicializar la aplicación Qt."""      app = QApplication.instance()  # Verifica si ya existe una instancia de QApplication      if not app:  # Si no existe, crea una nueva          app = QApplication([])      return app  @pytest.fixture  def button\_search(app):      """Fixture para inicializar Button\_Search."""      return Button\_Search(icon="utils/icons/search-512.png")  @pytest.fixture  def search\_bar(app):      """Fixture para inicializar Search\_Bar."""      return Search\_Bar()  ### Pruebas para Button\_Search ###  def test\_button\_search\_default\_text(button\_search):      """Verificar que el texto por defecto sea 'Buscar'."""      assert button\_search.text() == "Buscar"  def test\_button\_search\_signal\_emission(qtbot, button\_search):      """Verificar que la señal `signal\_presionado` se emita correctamente al presionar."""      with qtbot.wait\_signal(button\_search.signal\_presionado, timeout=1000):          qtbot.mouseClick(button\_search, Qt.LeftButton)  def test\_button\_search\_text\_update(button\_search):      """Verificar que se pueda actualizar dinámicamente el texto del botón."""      button\_search.setText("Nuevo Texto")      assert button\_search.text() == "Nuevo Texto", "El texto del botón no se actualizó correctamente."  ### Pruebas para Search\_Bar ###  def test\_search\_bar\_default\_text(search\_bar):      """Verificar que el texto predeterminado en la barra de búsqueda está vacío."""      assert search\_bar.text() == ""  def test\_search\_bar\_text\_update(search\_bar, qtbot):      """Probar si el texto ingresado en la barra de búsqueda se actualiza correctamente."""      qtbot.keyClicks(search\_bar, "Hola Mundo")      assert search\_bar.text() == "Hola Mundo"  def test\_search\_bar\_stylesheet(search\_bar):      """Verificar que el estilo CSS aplicado a la barra de búsqueda es el correcto."""      expected\_stylesheet = "border: 2px solid #f2784b;border-radius:5px;"      assert search\_bar.styleSheet() == expected\_stylesheet, "El estilo CSS no coincide con el esperado." |



# **7. Bibliografía**

* **Qt for Python. (s. f.). Doc.qt.io. Recuperado 21 de noviembre de 2024, de** <https://doc.qt.io/qtforpython-6/>
* **pytest-qt — pytest-qt documentation. (s. f.). Readthedocs.io. Recuperado 21 de noviembre de 2024, de** <https://pytest-qt.readthedocs.io/en/latest/>