

A thick dark blue vertical bar runs down the left side of the page. A purple arrow points to the right, overlapping the bar.

# Tema 03 – Boletín 01

Desarrollo de interfaces 2024/25

Several thin, curved lines in dark blue and light grey originate from the bottom left corner and sweep upwards and to the right.

Pedro Blanco Vargas  
Cristina Bermúdez Castro  
Noah Montaña Muñoz

---

# ÍNDICE

---

1.- Creación de los componentes personalizados.....	3
1.1.- Creación de la barra de búsqueda.....	3
1.1.1.- Definición y funcionalidad. ....	3
1.1.2.- Diseño visual.....	4
1.1.3.- Interacción y comportamiento.....	5
1.1.4.- Estructura y layout.....	5
1.2.- Creación del botón de confirmación.....	6
1.2.1.- Definición y funcionalidad. ....	6
1.2.2.- Configuración de tamaño. ....	7
1.2.3.- Carga de la imagen.....	7
1.2.4.- Estilo visual.....	7
2.- Importar los componentes personalizados a otro proyecto.....	8
2.1.- Estructura del paquete.....	8
2.2.- Archivo setup.py.....	9
2.3.- Archivo __init__.py. ....	10
2.4.- Empaquetado e instalación del paquete.....	10
3.- Uso del paquete en el proyecto principal. ....	11
3.1.- Importación del paquete y los componentes personalizados. ....	12
3.2.- Reemplazo de widgets estándar con componentes personalizados.....	12
4.- Gestión de la tabla de datos con búsqueda y restauración. ....	13
4.1.- Explicación del flujo para obtener y mostrar datos en la tabla. ....	13
4.2.- Obtención de datos desde la base de datos.....	13
4.2.1.- Definición y funcionalidad del controlador.....	13
4.3.1.- Configuración inicial de la tabla.....	14
4.4.- Implementación de la barra de búsqueda personalizada. ....	16

4.4.1.- Texto ingresado en el SearchBar. ....	16
4.4.2.- Manejo de la señal en la ventana principal (HomeWindow). ....	16
4.4.3.- Actualización de la tabla con los resultados. ....	17
4.5.- Funcionalidad del botón "Volver". ....	18
4.5.1.- Creación del botón personalizado. ....	18
4.5.2.- Conexión a la función on_back_button_clicked. ....	18
5.- Pruebas Unitarias. ....	18
5.1.- ¿Qué es una prueba unitaria? ....	18
5.2.- Importación de Módulos y Configuración Inicial. ....	19
5.3.- Clase TestSearchBar. ....	19
5.3.1.- Método setUp. ....	20
5.3.2.- Método tearDown. ....	20
5.3.3.- Método test_search_input. ....	20
5.3.4. Método test_input_text. ....	21
5.4. Bloque Principal. ....	22
5.5.- Clase test_search_button. ....	22
5.5.1.- Clase SearchButton. ....	22
5.5.2.- Clase de Pruebas TestSearchButton ....	23
5.5.3.- Método de Prueba test_search_button_click. ....	24
5.6.- Clase test_back_button. ....	27
5.6.1.- Clase BackButton ....	27
5.6.2.- Clase de Pruebas TestBackButton ....	28
5.6.3.- Método de Prueba test_back_button_click ....	28
6.- Despliegue de la Aplicación ....	31
6.1.- Entorno virtual ....	31
6.1.1.- Cómo crear el entorno virtual en el proyecto. ....	31
6.1.2.- Cómo activar el entorno virtual. ....	32
6.1.3.- ¿Qué es el archivo requirements.txt y qué contiene? ....	32
6.1.4.- Cómo instalar las dependencias desde el archivo requirements.txt ....	33
6.2.- Cómo convertir un archivo ui en un py. ....	34
6.3.- Docker y BBDD PostgreSQL. ....	34
6.3.1.- ¿Qué es Docker Compose y para que se utiliza? ....	34
6.3.2.- Levantar el Contenedor ....	35
6.4.- Conectarnos a la BBDD desde DBeaver. ....	35

## 1.- Creación de los componentes personalizados.

En el desarrollo de este boletín, nos enfocaremos en los componentes personalizados en Python, una herramienta clave para construir soluciones modulares, reutilizables y adaptadas a las necesidades específicas de un proyecto. Estos componentes permiten encapsular funcionalidades, mejorar la legibilidad del código y facilitar su mantenimiento.

A continuación, exploraremos los conceptos y pasos necesarios para diseñar y crear componentes personalizados que cumplan con estándares de calidad y optimicen el flujo de trabajo.

### 1.1.- Creación de la barra de búsqueda.

El componente **SearchBar** es un widget personalizado desarrollado utilizando **PySide6**, una biblioteca que permite construir interfaces gráficas (GUIs) en Python. Este componente se diseñó para actuar como una barra de búsqueda funcional y visualmente atractiva, ideal para integrarse en aplicaciones gráficas.

#### 1.1.1.- Definición y funcionalidad.

El componente hereda de la clase base **QWidget**, lo que le permite combinar otros widgets en su interior y actuar como una unidad independiente dentro de la interfaz gráfica:

```
class SearchBar(QWidget):
```

### Señales y atributos principales:

**Señal personalizada:** La señal **search\_submitted**, definida mediante **Signal**, permite comunicar el texto ingresado a otros componentes de la aplicación cuando se presiona la tecla Enter:

```
search_submitted = Signal(str)
```

**Campo de entrada de texto:** La barra de búsqueda está representada por un widget **QLineEdit**, que captura el texto del usuario y permite personalizar su apariencia y comportamiento:

```
self.search_input = QLineEdit(self)
```

### 1.1.2.- Diseño visual.

El diseño visual se define mediante hojas de estilo CSS aplicadas al campo de texto **self.search\_input**, para garantizar una apariencia moderna y profesional:

```
self.search_input.setStyleSheet("""
    QLineEdit {
        padding: 4px;
        font-family: "Arial";
        font-size: 14px;
        color: #4A3D2D;
        border: 1px solid rgba(0, 0, 0, 0.5);
        border-radius: 5px;
        background-color: rgba(255, 255, 255, 0.7);
    }
    QLineEdit:focus {
        border: 1px solid #C69C8D;
        background-color: rgba(255, 255, 255, 1);
    }
})""")
```

**Estilo base:** La fuente **Arial**, el tamaño de texto de 14px y los bordes redondeados hacen que el componente sea atractivo y fácil de leer.

**Interacción con el usuario:** El cambio de color del borde y el fondo al obtener el foco mejora la experiencia de usuario, indicando visualmente que el campo está activo.

### 1.1.3.- Interacción y comportamiento.

El método **on\_search** gestiona el evento cuando el usuario presiona la tecla Enter. Se conecta al evento **returnPressed** del campo de texto:

```
self.search_input.returnPressed.connect(self.on_search)
```

### Flujo del método:

1.- Captura el texto ingresado por el usuario y elimina los espacios en blanco al inicio y al final:

```
text = self.search_input.text().strip()
```

2.- Si el texto no está vacío, emite la señal **search\_submitted** con el contenido ingresado:

```
if text:  
    self.search_submitted.emit(text)
```

3.- Limpia el campo de texto para permitir nuevas búsquedas:

```
self.search_input.clear()
```

### 1.1.4.- Estructura y layout.

El diseño del componente se organiza utilizando un **QHBoxLayout**, que coloca el campo de texto en una disposición horizontal sin márgenes adicionales:

```
layout = QHBoxLayout()  
layout.addWidget(self.search_input)  
layout.setContentsMargins(0, 0, 0, 0)  
self.setLayout(layout)
```

Además, el tamaño total del widget **SearchBar** se define como 220x30 píxeles para garantizar un diseño compacto y consistente:

```
self.setFixedSize(220, 30)
```

## 1.2.- Creación del botón de confirmación.

El componente **ImageButton** es un botón personalizado desarrollado utilizando **PySide6**. Este botón está diseñado para utilizar imágenes como íconos en lugar de texto, lo que lo convierte en una opción ideal para interfaces gráficas modernas donde se prefiera un enfoque más visual. A continuación, se detalla la estructura y funcionamiento del componente.

### 1.2.1.- Definición y funcionalidad.

El componente hereda de **QPushButton**, una clase base en PySide6 utilizada para crear botones interactivos. Este diseño personalizado combina la funcionalidad estándar de un botón con la capacidad de mostrar imágenes de forma elegante.

```
class ImageButton(QPushButton):  
    def __init__(self, image_path, button_size=QSize(20, 20),  
parent=None):  
    super().__init__(parent)
```

### Parámetros clave:

- **image\_path**: Ruta a la imagen que se utilizará como ícono del botón.
- **button\_size**: Tamaño del botón (y del ícono). Se establece como un objeto **QSize**.
- **parent**: Widget principal al que pertenece el botón.

El constructor inicializa el botón, establece su tamaño, carga la imagen y aplica estilos personalizados.

### 1.2.2.- Configuración de tamaño.

El tamaño del botón se ajusta utilizando **setFixedSize**, lo que asegura que tanto el botón como el ícono mantengan las dimensiones definidas por el parámetro **button\_size**. Además, el tamaño del ícono se adapta al botón mediante **setIconSize**:

```
self.setFixedSize(button_size)
self.setIconSize(button_size)
```

### 1.2.3.- Carga de la imagen.

La imagen se convierte en un ícono mediante las clases **QIcon** y **QPixmap**, que permiten cargar y manejar imágenes de manera eficiente en PySide6. Esta imagen se asigna al botón utilizando:

```
self.setIcon(QIcon(QPixmap(image_path)))
```

Esto asegura que la imagen se renderice correctamente como parte del botón.

### 1.2.4.- Estilo visual.

El estilo del botón se define mediante **hojas de estilo CSS**, logrando una apariencia limpia y funcional:

```
self.setStyleSheet("""
    QPushButton {
        border: none;
        background-color: transparent;
    }
    QPushButton:hover {
        background-color: rgba(0, 0, 0, 0.05);
        border-radius: 12px;
    }
    QPushButton:pressed {
        background-color: rgba(0, 0, 0, 0.1);
        border-radius: 12px;
    }
    """)
```



### Detalles del estilo:

- **Sin bordes ni fondo inicial:** El botón es completamente transparente, mostrando solo el ícono.
- **Efecto *hover*:** Cambia ligeramente el color de fondo al pasar el cursor sobre el botón, mejorando la interacción visual.
- **Efecto *pressed*:** Un cambio más pronunciado en el color de fondo y un borde redondeado indican que el botón está siendo presionado.

## 2.- Importar los componentes personalizados a otro proyecto.

Para poder reutilizar los componentes personalizados **SearchBar** e **ImageButton** en otros proyectos, es necesario empaquetarlos correctamente. Este proceso implica organizar los archivos, configurarlos como un paquete Python y asegurarse de que sean fácilmente instalables e importables en cualquier entorno. A continuación, se detalla cómo empaquetar estos componentes utilizando los archivos **setup.py** y **\_\_init\_\_.py**, y cómo integrarlos en otro proyecto.

### 2.1.- Estructura del paquete.

Antes de empaquetar, los archivos deben organizarse con una estructura clara, ya que luego a la hora de importar las clases en nuestro proyecto principal, esto nos ahorrará varios problemas. La estructura que hemos seguido es la siguiente:

```
Componentes_EQ05 /  
├── Componentes_EQ05  
│   ├── __init__.py  
│   ├── image_button.py  
│   └── search_bar.py  
└── setup.py
```



```
▼ COMPONENTES_EQ05  
▼ Componentes_EQ05  
  > __pycache__  
  > Componentes_EQ05.egg-info  
  > dist  
  > venv  
  + __init__.py  
  + image_button.py  
  + search_bar.py  
  > Componentes_EQ05.egg-info  
  > dist  
  + setup.py
```

- **Componentes\_EQ05/:** Carpeta del proyecto
- **Componentes\_EQ05:** Carpeta con el mismo nombre que el proyecto para luego poder importar las clases de forma más cómoda. Contiene los módulos con los componentes personalizados.
- **\_\_init\_\_.py:** Archivo que define los elementos del paquete que estarán disponibles al importarlo.
- **setup.py:** Archivo que configura y describe el paquete para su distribución. Deberá estar en la raíz del proyecto.

## 2.2.- Archivo setup.py.

El archivo **setup.py** utiliza **setuptools** para definir las configuraciones básicas del paquete. Mostramos a continuación el código de esta clase:

```
from setuptools import setup, find_packages

setup(
    name="Componentes_EQ05",
    version="1.0",
    description="Componentes personalizados con PySide6",
    author="Equipo_05",
    packages=find_packages(),
    install_requires=["PySide6"],
)
```

### Detalles clave:

- **name:** Nombre del paquete, que será utilizado al instalarlo (por ejemplo, con pip).
- **version:** Versión del paquete. Es importante mantener un control de versiones para actualizaciones.
- **packages=find\_packages():** Detecta automáticamente los submódulos (en este caso, la carpeta componentes).
- **install\_requires:** Lista de dependencias requeridas, como PySide6.

### 2.3.- Archivo `__init__.py`.

El archivo `__init__.py` define qué elementos estarán disponibles al importar el paquete. En este caso, incluye los dos componentes:

```
from .image_button import ImageButton
from .search_bar import SearchBar
```

Esto permite que ambos componentes puedan importarse directamente desde el paquete sin necesidad de especificar los archivos individuales. ¿Cómo se haría?:

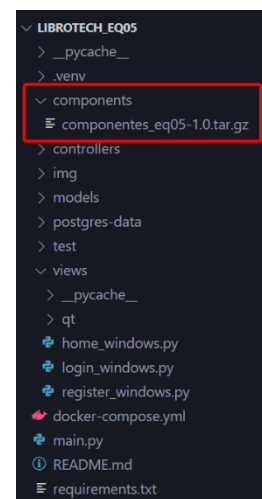
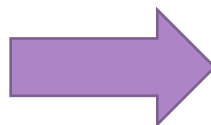
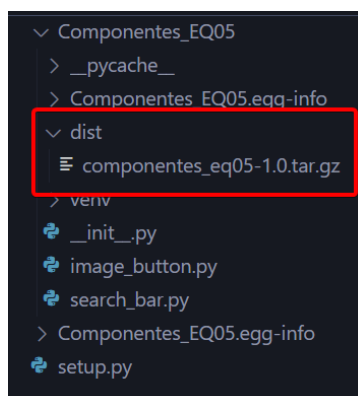
```
from Componentes_EQ05 import SearchButton, SearchBar, BackButton
```

### 2.4.- Empaquetado e instalación del paquete

Para empaquetar el proyecto, primeramente, navegaremos hacia la carpeta donde está el archivo `setup.py` (En la raíz del proyecto). Una vez colocados, generaremos el archivo del paquete lanzando el siguiente comando para crear el **comprimido instalable**:

➤ `python setup.py sdist`

Esto generará un directorio **dist/** que contendrá un archivo llamado **componentes\_eq05-1.0.tar.gz**. Este paquete lo moveremos hacia nuestro proyecto principal en una carpeta donde iremos dejando los paquetes que queramos meter en nuestro proyecto.



Ahora, antes de instalar el paquete, configuraremos el **requirements** del proyecto principal y es que, si queremos que este paquete se instale cuando creamos el entorno y instalamos las dependencias, deberemos indicar en el archivo requirements con una **ruta relativa** donde se encuentra este paquete. Para esto simplemente agregaremos al principio del archivo la siguiente línea en el **requirements.txt**:

```
Componentes_EQ05 @ file:../components/componentes_eq05-1.0.tar.gz  
psycpg==3.2.1  
psycpg-binary==3.2.1  
psycpg-pool==3.2.2  
psycpg2==2.9.10  
PySide6==6.7.2  
PySide6_Addons==6.7.2  
PySide6_Essentials==6.7.2  
setuptools==72.1.0  
shiboken6==6.7.2  
typing_extensions==4.12.2  
tzdata==2024.1  
wheel==0.44.0
```

Ahora, cuando instalemos las dependencias usando el archivo requirements.txt, se instalarán **automáticamente** los componentes personalizados que hemos creado anteriormente.

Otra forma más manual de instalar el paquete en nuestro proyecto es instalarlo con “**pip install**” y la ruta hacia el paquete.

➤ `pip install ../components/componentes_eq05-1.0.tar.gz`

Tras la instalación de la forma que sea, el paquete estará **disponible** para ser importado.

### 3.- Uso del paquete en el proyecto principal.

En este punto, se explica cómo se importaron y utilizaron los componentes personalizados `SearchBar`, `SearchButton` y `BackButton` dentro de la clase `HomeWindow` del proyecto principal.

### 3.1.- Importación del paquete y los componentes personalizados.

En el archivo del proyecto principal, los componentes personalizados fueron importados desde el paquete previamente empaquetado e instalado, llamado Componentes\_EQ05. Para importarlos simplemente escribiremos la siguiente línea:

```
from Componentes_EQ05 import SearchButton, SearchBar, BackButton
```

#### Explicación:

- **SearchButton:** Botón con una imagen personalizada.
- **SearchBar:** Barra de búsqueda interactiva.
- **BackButton:** Botón para volver a un estado o vista anterior.

Estas clases están disponibles gracias al empaquetado del proyecto y su instalación mediante el archivo setup.py.

### 3.2.- Reemplazo de widgets estándar con componentes personalizados.

Para la colocación de los componentes en nuestra nueva ventana, hemos dejado creado el input y los botones en la interfaz desde **QtDesigner**, de manera que luego en el código sustituimos estos elementos base por los componentes personalizados haciendo uso del método **replaceWidget**.

```
# Reemplazar el botón estándar con el botón personalizado
self.custom_button = SearchButton("img/prueba/buscar.png")
self.ui.horizontalLayout_4.replaceWidget(self.ui.pushButton,
self.custom_button)
self.ui.pushButton.deleteLater() # Eliminar el widget antiguo
self.custom_button.clicked.connect(self.on_search_button_clicked)

# barra de búsqueda personalizada
self.search_bar = SearchBar()
self.ui.horizontalLayout_4.replaceWidget(self.ui.lineEdit,
self.search_bar)
self.ui.lineEdit.deleteLater() # Eliminar el widget antiguo
self.search_bar.search_submitted.connect(self.on_search)
```

```
# Botón volver
self.custom_buttonBack = BackButton("img/prueba/back.png")
self.ui.horizontalLayout_4.replaceWidget(self.ui.pushButton_2,
self.custom_buttonBack)
self.ui.pushButton_2.deleteLater() # Eliminar el widget
antiguo
self.custom_buttonBack.clicked.connect(self.on_back_button_clicked)
```

## Detalles

1. Instanciamos los componentes
2. Sustituimos el antiguo widget por el nuevo
3. Eliminamos el antiguo widget
4. Le damos la funcionalidad al componente conectándolo a un evento y pasándole un método.

## 4.- Gestión de la tabla de datos con búsqueda y restauración.

### 4.1.- Explicación del flujo para obtener y mostrar datos en la tabla.

1. **Carga inicial:**
  - Cuando la ventana se inicia, llama a **cargar\_libros()** para llenar la tabla con todos los libros desde la base de datos.
2. **Búsqueda:**
  - El usuario ingresa texto en el **SearchBar** y activa la búsqueda.
  - Se filtran los datos obtenidos de la base de datos según el texto ingresado.
  - Los resultados filtrados se muestran en la tabla.
3. **Volver:**
  - Al presionar el botón "volver", simplemente se vuelve a llamar a **cargar\_libros()** para mostrar todos los libros nuevamente.

### 4.2.- Obtención de datos desde la base de datos.

#### 4.2.1.- Definición y funcionalidad del controlador.

El proceso comienza en el controlador de la base de datos, que gestiona la interacción entre la aplicación y la base de datos. En nuestra aplicación el controlador es **LibroController**, el cual tiene métodos que interactúan con la base de datos para obtener datos.

Por ejemplo, realiza una consulta a la base de datos, y devuelve los datos en forma de lista de objetos.

```
class LibroController:
    """
    Controlador que maneja las interacciones con los objetos libros
    """

    def __init__(self):
        # Instancia del CRUD para realizar operaciones con la base de
        # datos
        print("iniciando controller libro")
        self.crud_libro = CRUDLibro()
        # __init__

    def obtener_libros(self):

        print("He llegado al controller de obtener libro.") # Mensaje
        # de error

        """
        Método para obtener los libros existentes en la base de datos
        """
        libros = self.crud_libro.obtener_libros() # Obtener los libros
        # de la base de datos

        return libros # Retornar libros
```

#### 4.3.- Carga de los datos en la tabla.

##### 4.3.1.- Configuración inicial de la tabla.

Una vez obtenida la lista de libros, se recorre y se agregan filas al modelo de la tabla (QStandardItemModel).

```
def cargar_libros(self):
    """Obtiene los libros de la base de datos y los muestra en el
    listView."""

    print("He llegado a cargar libro.") # Mensaje de error

    try:

        # Crear y configurar el modelo
        model = QStandardItemModel()
        model.setHorizontalHeaderLabels(["Isbn", "Título", "Autor",
        "Género", "Publicación"])
        self.libro_controller = LibroController()
        libros = self.libro_controller.obtener_libros()
```

```
#####
# Configurar el modelo en el QTableView
self.ui.tableView.setModel(model)

# Configurar para que las filas se adapten al espacio
disponible
vertical_header = self.ui.tableView.verticalHeader()
vertical_header.setSectionResizeMode(QtWidgets.QHeaderView.
Stretch) # Las filas se estiran proporcionalmente

# Ajustar columnas
header = self.ui.tableView.horizontalHeader()
header.setSectionResizeMode(QtWidgets.QHeaderView.Stretch)

# Ajustar filas
vertical_header = self.ui.tableView.verticalHeader()
vertical_header.setSectionResizeMode(QtWidgets.QHeaderView.
Stretch)

# Ajustar columnas al contenido inicial
self.ui.tableView.resizeColumnsToContents()

# Configurar para que las columnas se adapten al espacio
disponible
header = self.ui.tableView.horizontalHeader()
header.setSectionResizeMode(QtWidgets.QHeaderView.Stretch)
# Todas las columnas se estiran proporcionalmente
#####

for libro in libros:
    fila = [
        QTableWidgetItem(str(libro.isbn)),
        QTableWidgetItem(libro.titulo),
        QTableWidgetItem(libro.autor),
        QTableWidgetItem(str(libro.genero)),
        QTableWidgetItem(str(libro.publicacion))
    ]
    for item in fila:
        item.setTextAlignment(Qt.AlignCenter) # Alinear
contenido
        model.appendRow(fila)

# Configurar el modelo en listView_libros
self.ui.tableView.setModel(model)

finally:
    print("OBTENIDOS LIBROS CORRECTAMENTE")
```



Este proceso se asegura de que cada registro se convierta en una fila en el **QTableView**.

#### 4.4.- Implementación de la barra de búsqueda personalizada.

El **SearchBar** permite al usuario filtrar libros por su título.

##### 4.4.1.- Texto ingresado en el SearchBar.

Cuando el usuario escribe un texto y presiona la tecla Enter o hace clic en el botón de búsqueda, se emite una señal **search\_submitted** con el texto como parámetro.

```
class SearchBar(QWidget):  
    # Señal personalizada que se emite cuando se realiza una búsqueda  
    search_submitted = Signal(str)  
  
    def on_search(self):  
        text = self.search_input.text().strip()  
        if text:  
            self.search_submitted.emit(text)  
            self.search_input.clear()
```

##### 4.4.2.- Manejo de la señal en la ventana principal (HomeWindow).

La señal se conecta al método **on\_search**, que filtra los datos y actualiza la tabla.

Conexión:

```
self.search_bar.search_submitted.connect(self.on_search)
```

#### Método:

```
def on_search(self, search_text):  
    """  
        Filtra los libros en la tabla en función del texto ingresado en  
        la barra de búsqueda.  
    """  
    print(f"Buscando: {search_text}")  
  
    # Obtener todos los libros desde el controlador  
    libros = self.libro_controller.obtener_libros()  
  
    # Filtrar los libros por el texto ingresado  
    libros_filtrados = [  
        libro for libro in libros
```

```
        if search_text.lower() in libro.titulo.lower(): #  
Coincidencia insensible a mayúsculas  
    ]  
  
    # Actualizar la tabla con los libros filtrados  
    self.mostrar_libros(libros_filtrados)
```

#### 4.4.3.- Actualización de la tabla con los resultados.

Se vuelve a construir el modelo de la tabla usando solo los libros que coinciden con la búsqueda y se muestra el resultado.

```
def mostrar_libros(self, libros):  
    """  
    Muestra una lista de libros en el QTableView.  
    """  
    model = QStandardItemModel()  
    model.setHorizontalHeaderLabels(["Isbn", "Título", "Autor",  
"Género", "Publicación"])  
  
    for libro in libros:  
        fila = [  
            QStandardItem(str(libro.isbn)),  
            QStandardItem(libro.titulo),  
            QStandardItem(libro.autor),  
            QStandardItem(libro.genero),  
            QStandardItem(str(libro.publicacion))  
        ]  
        for item in fila:  
            item.setTextAlignment(Qt.AlignCenter) # Alinear  
contenido al centro  
        model.appendRow(fila)  
  
    # Configurar el modelo en la tabla  
    self.ui.tableView.setModel(model)  
  
    # Ajustar las columnas automáticamente  
    self.ui.tableView.horizontalHeader().setSectionResizeMode(QHead  
erView.Stretch)  
  
    # Ajustar la altura de las filas al contenido del texto  
    self.ui.tableView.verticalHeader().setSectionResizeMode(QHeader  
View.ResizeToContents)
```

#### 4.5.- Funcionalidad del botón "Volver".

El botón "Volver" sirve para restaurar la lista completa de libros en la tabla, después de que el usuario haya realizado una búsqueda.

##### 4.5.1.- Creación del botón personalizado.

En el método donde se configuran los elementos de la interfaz (**HomeWindow**), se reemplaza el botón estándar con un botón personalizado (**BackButton**).

```
# Boton volver
self.custom_buttonBack = BackButton("img/prueba/back.png")
self.ui.horizontalLayout_4.replaceWidget(self.ui.pushButton_2,
self.custom_buttonBack)
self.ui.pushButton_2.deleteLater() # Eliminar el widget
antiguo
self.custom_buttonBack.clicked.connect(self.on_back_button_clicked)
```

**custom\_buttonBack** es el botón que usa el usuario para volver a la tabla completa.

##### 4.5.2.- Conexión a la función on\_back\_button\_clicked.

Cuando se hace clic en el botón "Volver", se ejecuta el método **on\_back\_button\_clicked**, que se encarga de restaurar todos los datos en la tabla.

```
def on_back_button_clicked(self):
    """
    Recarga la lista completa de libros cuando se hace clic en el
    botón "Volver".
    """
    self.cargar_libros() # Recargar todos los libros
```

## 5.- Pruebas Unitarias.

### 5.1.- ¿Qué es una prueba unitaria?

Una prueba unitaria es una técnica de desarrollo de software que consiste en probar de manera aislada pequeñas partes del código (generalmente funciones, métodos o clases) para asegurarse de que funcionan correctamente. En Python, el módulo `unittest` proporciona una herramienta estándar para escribir y ejecutar estas pruebas.

## 5.2.- Importación de Módulos y Configuración Inicial.

Se **importarán** las clases y las funciones:

- **unittest** para escribir y ejecutar pruebas unitarias.
- **QApplication** y **QLineEdit** de PySide6 para manejar widgets de la interfaz gráfica.
- **SearchBar** desde el módulo Componentes\_EQ05, que es el componente que se va a probar.

### Instancia única de QApplication:

Se crea una única instancia global de **QApplication**, necesaria para inicializar cualquier widget de **PySide6**. Esto evita errores al ejecutar múltiples pruebas.

```
import unittest
from PySide6.QtWidgets import QApplication, QLineEdit
from Componentes_EQ05 import SearchBar # Asegúrate de que el import sea correcto
```

## 5.3.- Clase TestSearchBar.

La clase hereda de **unittest.TestCase** y define los métodos para realizar pruebas sobre el componente **SearchBar**.

```
class TestSearchBar(unittest.TestCase):
    def setUp(self):
        """Configurar el entorno para las pruebas"""
        self.search_bar = SearchBar() # Instanciar el componente a probar

    def tearDown(self):
        """Limpiar después de cada prueba"""
        self.search_bar.deleteLater() # Eliminar el widget para evitar fugas de memoria

    def test_search_input(self):
        """Prueba que la búsqueda emite la señal correctamente"""
        # Crear un 'spy' para conectar con la señal
        result = []

        def spy(value):
            result.append(value)

        self.search_bar.search_submitted.connect(spy) # Conectar la señal
        self.search_bar.search_submitted.emit("Don Quijote") # Emitir la señal

        self.assertEqual(result, ["Don Quijote"]) # Verificar que la señal envió el valor esperado

    def test_input_text(self):
        """Probar que el texto introducido es correcto"""
        self.search_bar.search_input.setText("Don Quijote")
        self.assertEqual(self.search_bar.search_input.text(), "Don Quijote") # Verificar el texto

if __name__ == "__main__":
    unittest.main()
```

### 5.3.1.- Método setUp.

**Propósito:**

Configurar el entorno para cada prueba.

**Qué hace:**

Crea una nueva instancia de **SearchBar** antes de cada prueba, asegurándose de que cada prueba tenga un estado limpio.

```
def setUp(self):  
    """Configurar el entorno para las pruebas"""  
    self.search_bar = SearchBar() # Instanciar el componente a probar
```

### 5.3.2.- Método tearDown.

**Propósito:**

Limpiar el entorno después de cada prueba.

**Qué hace:**

Libera los recursos asociados al widget **SearchBar** mediante **deleteLater()** para evitar fugas de memoria o referencias innecesarias.

```
def tearDown(self):  
    """Limpiar después de cada prueba"""  
    self.search_bar.deleteLater() # Eliminar el widget para evitar fugas de memoria
```

### 5.3.3.- Método test\_search\_input.

**Propósito:**

Verificar que la señal **search\_submitted** se emite correctamente cuando es invocada con un valor.

**Qué hace:**

1. **Crea un 'spy':**

- Se define una función spy para capturar los valores emitidos por la señal.

**Conecta el spy a la señal:**

- `self.search_bar.search_submitted.connect(spy)` conecta la señal `search_submitted` al spy.

**2. Emite la señal:**

- self.search\_bar.search\_submitted.emit("Don Quijote") emite la señal con el valor "Don Quijote".

**3. Verifica el resultado:**

- Comprueba que el spy capturó correctamente el valor emitido, usando self.assertEqual(result, ["Don Quijote"]).

```
def test_search_input(self):
    """Prueba que la búsqueda emite la señal correctamente"""
    # Crear un 'spy' para conectar con la señal
    result = []

    def spy(value):
        result.append(value)

    self.search_bar.search_submitted.connect(spy) # Conectar la señal
    self.search_bar.search_submitted.emit("Don Quijote") # Emitir la señal

    self.assertEqual(result, ["Don Quijote"]) # Verificar que la señal envió el valor esperado
```

**5.3.4. Método test\_input\_text.****Propósito:**

- Verificar que el texto introducido en el campo de entrada (**QLineEdit**) es manejado correctamente.

**Qué hace:****1. Establece el texto:**

- self.search\_bar.search\_input.setText("Don Quijote") configura el texto del campo de entrada.

**2. Verifica el texto:**

- Comprueba que el texto del campo coincide con lo que se configuró, usando self.assertEqual(self.search\_bar.search\_input.text(), "Don Quijote").

```
def test_input_text(self):
    """Probar que el texto introducido es correcto"""
    self.search_bar.search_input.setText("Don Quijote")
    self.assertEqual(self.search_bar.search_input.text(), "Don Quijote") # Verificar el texto
```

## 5.4. Bloque Principal.

### Propósito:

- Ejecutar las pruebas si el archivo se ejecuta directamente.

### Qué hace:

- Llama a **unittest.main()** para buscar y ejecutar todas las pruebas definidas en la clase.

```
if __name__ == "__main__":  
    unittest.main()
```

## Resumen de las Funcionalidades Probadas

### Señal **search\_submitted**:

- Asegura que la señal envía los datos esperados cuando es activada.

### Campo de entrada de texto:

- Verifica que el texto ingresado se almacena y se puede recuperar correctamente.

```
(.venv) PS C:\Users\Noah\Desktop\LibroTech_Eq05> python -m unittest test.test_search_bar  
..  
-----  
Ran 2 tests in 0.009s  
..  
-----  
Ran 2 tests in 0.009s
```

## 5.5.- Clase **test\_search\_button**.

### 5.5.1.- Clase **SearchButton**.

La clase **SearchButton** hereda de **QWidget** y define una interfaz con un campo de entrada de texto (**search\_input**) y un botón (**search\_button**). Esta clase tiene una señal personalizada llamada **search\_submitted**, que se emite cuando el usuario hace clic en el botón de búsqueda.

- **search\_submitted**: Señal que emite el texto ingresado en el campo de texto cuando se hace clic en el botón de búsqueda.
- **Método emit\_search**: Conecta el clic del botón con la emisión de la señal, enviando el texto del campo de entrada.

```
class SearchButton(QWidget):
    search_submitted = Signal(str) # Señal que emite el texto cuando se hace clic en el botón

    def __init__(self):
        super().__init__()

        self.search_input = QLineEdit(self) # Campo de entrada
        self.search_button = QPushButton("Buscar", self) # Botón de búsqueda
        self.search_button.clicked.connect(self.emit_search) # Conectar el clic del botón a la función

        layout = QVBoxLayout(self)
        layout.addWidget(self.search_input)
        layout.addWidget(self.search_button)

    def emit_search(self):
        """Emite la señal con el texto del campo de entrada cuando se hace clic en el botón"""
        self.search_submitted.emit(self.search_input.text())
```

### 5.5.2.- Clase de Pruebas TestSearchButton

Esta clase define la prueba unitaria para el botón de búsqueda. Utiliza **unittest** para ejecutar la prueba.

- **setUp**: Método que se ejecuta antes de cada prueba. Instancia el widget SearchButton, creando el entorno necesario para la prueba.
- **tearDown**: Método que se ejecuta después de cada prueba. Elimina el widget SearchButton para evitar fugas de memoria.

```
class TestSearchButton(unittest.TestCase):
    def setUp(self):
        """Configurar el entorno para las pruebas"""
        self.search_button_widget = SearchButton() # Instanciar el widget que contiene el botón de búsqueda

    def tearDown(self):
        """Limpiar después de cada prueba"""
        self.search_button_widget.deleteLater() # Eliminar el widget para evitar fugas de memoria

    def test_search_button_click(self):
        """Probar que al hacer clic en el botón de búsqueda se emite la señal correctamente"""
        # Crear un 'spy' para capturar la señal
        result = []

        def spy(value):
            result.append(value)

        # Conectar la señal search_submitted al spy
        self.search_button_widget.search_submitted.connect(spy)

        # Establecer un texto en el campo de entrada
        self.search_button_widget.search_input.setText("Don Quijote")

        # Hacer clic en el botón de búsqueda
        self.search_button_widget.search_button.click()

        # Verificar que la señal fue emitida con el valor correcto
        self.assertEqual(result, ["Don Quijote"])
```



### 5.5.3.- Método de Prueba test\_search\_button\_click

**Propósito:**

Verificar que, al hacer clic en el botón de búsqueda, la señal **search\_submitted** se emita correctamente con el texto del campo de entrada.

**Pasos:**

- **Crear un 'spy':** Se define una función llamada spy que captura el valor emitido por la señal search\_submitted.

```
class TestSearchButton(unittest.TestCase):
    def setUp(self):
        """Configurar el entorno para las pruebas"""
        self.search_button_widget = SearchButton() # Instanciar el widget que contiene el botón de búsqueda

    def tearDown(self):
        """Limpiar después de cada prueba"""
        self.search_button_widget.deleteLater() # Eliminar el widget para evitar fugas de memoria

    def test_search_button_click(self):
        """Probar que al hacer clic en el botón de búsqueda se emite la señal correctamente"""
        # Crear un 'spy' para capturar la señal
        result = []

        def spy(value):
            result.append(value)

        # Conectar la señal search_submitted al spy
        self.search_button_widget.search_submitted.connect(spy)

        # Establecer un texto en el campo de entrada
        self.search_button_widget.search_input.setText("Don Quijote")

        # Hacer clic en el botón de búsqueda
        self.search_button_widget.search_button.click()

        # Verificar que la señal fue emitida con el valor correcto
        self.assertEqual(result, ["Don Quijote"])
```

- **Conectar la señal al 'spy':** La señal search\_submitted se conecta a la función spy para capturar el valor emitido.

```
class TestSearchButton(unittest.TestCase):
    def setUp(self):
        """Configurar el entorno para las pruebas"""
        self.search_button_widget = SearchButton() # Instanciar el widget que contiene el botón de búsqueda

    def tearDown(self):
        """Limpiar después de cada prueba"""
        self.search_button_widget.deleteLater() # Eliminar el widget para evitar fugas de memoria

    def test_search_button_click(self):
        """Probar que al hacer clic en el botón de búsqueda se emite la señal correctamente"""
        # Crear un 'spy' para capturar la señal
        result = []

        def spy(value):
            result.append(value)

        # Conectar la señal search_submitted al spy
        self.search_button_widget.search_submitted.connect(spy)

        # Establecer un texto en el campo de entrada
        self.search_button_widget.search_input.setText("Don Quijote")

        # Hacer clic en el botón de búsqueda
        self.search_button_widget.search_button.click()

        # Verificar que la señal fue emitida con el valor correcto
        self.assertEqual(result, ["Don Quijote"])
```

- **Establecer el Texto:** Se configura el texto del campo de entrada (`search_input`) a "Don Quijote".

```
class TestSearchButton(unittest.TestCase):
    def setUp(self):
        """Configurar el entorno para las pruebas"""
        self.search_button_widget = SearchButton() # Instanciar el widget que contiene el botón de búsqueda

    def tearDown(self):
        """Limpiar después de cada prueba"""
        self.search_button_widget.deleteLater() # Eliminar el widget para evitar fugas de memoria

    def test_search_button_click(self):
        """Probar que al hacer clic en el botón de búsqueda se emite la señal correctamente"""
        # Crear un 'spy' para capturar la señal
        result = []

        def spy(value):
            result.append(value)

        # Conectar la señal search_submitted al spy
        self.search_button_widget.search_submitted.connect(spy)

        # Establecer un texto en el campo de entrada
        self.search_button_widget.search_input.setText("Don Quijote")

        # Hacer clic en el botón de búsqueda
        self.search_button_widget.search_button.click()

        # Verificar que la señal fue emitida con el valor correcto
        self.assertEqual(result, ["Don Quijote"])
```

- **Simular Clic:** Se simula un clic en el botón de búsqueda usando `search_button.click()`.

```
class TestSearchButton(unittest.TestCase):
    def setUp(self):
        """Configurar el entorno para las pruebas"""
        self.search_button_widget = SearchButton() # Instanciar el widget que contiene el botón de búsqueda

    def tearDown(self):
        """Limpiar después de cada prueba"""
        self.search_button_widget.deleteLater() # Eliminar el widget para evitar fugas de memoria

    def test_search_button_click(self):
        """Probar que al hacer clic en el botón de búsqueda se emite la señal correctamente"""
        # Crear un 'spy' para capturar la señal
        result = []

        def spy(value):
            result.append(value)

        # Conectar la señal search_submitted al spy
        self.search_button_widget.search_submitted.connect(spy)

        # Establecer un texto en el campo de entrada
        self.search_button_widget.search_input.setText("Don Quijote")

        # Hacer clic en el botón de búsqueda
        self.search_button_widget.search_button.click()

        # Verificar que la señal fue emitida con el valor correcto
        self.assertEqual(result, ["Don Quijote"])
```

- **Verificación:** Se comprueba que la señal fue emitida con el texto correcto mediante la comparación del valor capturado por el "spy" con el valor esperado.

```
class TestSearchButton(unittest.TestCase):
    def setUp(self):
        """Configurar el entorno para las pruebas"""
        self.search_button_widget = SearchButton() # Instanciar el widget que contiene el botón de búsqueda

    def tearDown(self):
        """Limpiar después de cada prueba"""
        self.search_button_widget.deleteLater() # Eliminar el widget para evitar fugas de memoria

    def test_search_button_click(self):
        """Probar que al hacer clic en el botón de búsqueda se emite la señal correctamente"""
        # Crear un 'spy' para capturar la señal
        result = []

        def spy(value):
            result.append(value)

        # Conectar la señal search_submitted al spy
        self.search_button_widget.search_submitted.connect(spy)

        # Establecer un texto en el campo de entrada
        self.search_button_widget.search_input.setText("Don Quijote")

        # Hacer clic en el botón de búsqueda
        self.search_button_widget.search_button.click()

        # Verificar que la señal fue emitida con el valor correcto
        self.assertEqual(result, ["Don Quijote"])
```

## Resumen de las Funcionalidades Probadas

### **Emisión de la Señal Correcta:**

- Se verifica que, al hacer clic en el botón de búsqueda, se emita la señal **search\_submitted** con el texto que el usuario ha ingresado en el campo de entrada (**search\_input**).

### **Interacción del Usuario:**

- La prueba simula la acción del usuario al ingresar texto en el campo de búsqueda y hacer clic en el botón de búsqueda, lo que **desencadena** la emisión de la señal.

### **Captura del Valor Emitido:**

- Se utiliza un **"spy"** (función de captura) para verificar que el valor emitido por la señal sea el texto esperado ("Don Quijote" en este caso), asegurando que el componente de búsqueda está pasando correctamente los datos.

```
(.venv) PS C:\Users\Noah\Desktop\LibroTech_Eq05> python -m unittest test.test_search_button
.
-----
Ran 1 test in 0.009s

OK
```

## 5.6.- Clase test\_back\_button

### 5.6.1.- Clase BackButton

La clase BackButton es un componente de la interfaz de usuario que contiene un **botón de retroceso** (QPushButton). Además, define una **señal personalizada** (back\_pressed) que se emite cuando el botón es presionado.

- **back\_pressed**: Es una señal personalizada que se emite al hacer clic en el botón de retroceso.
- **Método emit\_back**: Este método se conecta al clic del botón. Cuando el botón es presionado, emite la señal back\_pressed.

```
# Clase que contiene el botón de retroceso
class BackButton(QWidget):
    back_pressed = Signal() # Señal que emite cuando se presiona el botón de retroceso

    def __init__(self):
        super().__init__()

        self.back_button = QPushButton("Volver", self) # Botón de retroceso
        self.back_button.clicked.connect(self.emit_back) # Conectar el clic del botón a la función

    def emit_back(self):
        """Emite la señal de retroceso cuando se hace clic en el botón"""
        self.back_pressed.emit()
```

### 5.6.2.- Clase de Pruebas TestBackButton

Esta clase define la prueba unitaria que verificará el comportamiento del botón de retroceso.

- **setUp**: Este método se ejecuta antes de cada prueba. En él se instancia el widget BackButton, que es el objeto a probar.
- **tearDown**: Este método se ejecuta después de cada prueba para asegurarse de que el entorno esté limpio. En este caso, elimina el widget BackButton para evitar fugas de memoria.

```
# Clase que contiene las pruebas unitarias
class TestBackButton(unittest.TestCase):
    def setUp(self):
        """Configurar el entorno para las pruebas"""
        self.back_button_widget = BackButton() # Instanciar el widget que contiene el botón de retroceso

    def tearDown(self):
        """Limpiar después de cada prueba"""
        self.back_button_widget.deleteLater() # Eliminar el widget para evitar fugas de memoria

    def test_back_button_click(self):
        """Probar que al hacer clic en el botón de retroceso se emite la señal correctamente"""
        # Crear un 'spy' para capturar la señal
        result = []

        def spy():
            result.append("Retroceso")

        # Conectar la señal back_pressed al spy
        self.back_button_widget.back_pressed.connect(spy)

        # Hacer clic en el botón de retroceso
        self.back_button_widget.back_button.click()

        # Verificar que la señal fue emitida
        self.assertEqual(result, ["Retroceso"])
```

### 5.6.3.- Método de Prueba test\_back\_button\_click

Este método prueba que, al hacer clic en el botón de retroceso, se emita correctamente la señal back\_pressed.

- **Crear un 'spy'**: Se define una función spy que actúa como un espía para capturar cuando la señal back\_pressed es emitida.

```
def test_back_button_click(self):
    """Probar que al hacer clic en el botón de retroceso se emite la señal correctamente"""
    # Crear un 'spy' para capturar la señal
    result = []

    def spy():
        result.append("Retroceso")

    # Conectar la señal back_pressed al spy
    self.back_button_widget.back_pressed.connect(spy)

    # Hacer clic en el botón de retroceso
    self.back_button_widget.back_button.click()

    # Verificar que la señal fue emitida
    self.assertEqual(result, ["Retroceso"])
```

- **Conectar la señal al 'spy':** La señal back\_pressed del widget BackButton se conecta a la función spy para que esta se ejecute cada vez que la señal sea emitida.

```
def test_back_button_click(self):
    """Probar que al hacer clic en el botón de retroceso se emite la señal correctamente"""
    # Crear un 'spy' para capturar la señal
    result = []

    def spy():
        result.append("Retroceso")

    # Conectar la señal back_pressed al spy
    self.back_button_widget.back_pressed.connect(spy)

    # Hacer clic en el botón de retroceso
    self.back_button_widget.back_button.click()

    # Verificar que la señal fue emitida
    self.assertEqual(result, ["Retroceso"])
```

- **Simular Clic:** Se simula un clic en el botón de retroceso usando self.back\_button\_widget.back\_button.click().

```
def test_back_button_click(self):
    """Probar que al hacer clic en el botón de retroceso se emite la señal correctamente"""
    # Crear un 'spy' para capturar la señal
    result = []

    def spy():
        result.append("Retroceso")

    # Conectar la señal back_pressed al spy
    self.back_button_widget.back_pressed.connect(spy)

    # Hacer clic en el botón de retroceso
    self.back_button_widget.back_button.click()

    # Verificar que la señal fue emitida
    self.assertEqual(result, ["Retroceso"])
```

- **Verificación:** Finalmente, se verifica si el "espía" (spy) ha capturado correctamente la señal mediante `self.assertEqual(result, ["Retroceso"])`.

```
def test_back_button_click(self):
    """Probar que al hacer clic en el botón de retroceso se emite la señal correctamente"""
    # Crear un 'spy' para capturar la señal
    result = []

    def spy():
        result.append("Retroceso")

    # Conectar la señal back_pressed al spy
    self.back_button_widget.back_pressed.connect(spy)

    # Hacer clic en el botón de retroceso
    self.back_button_widget.back_button.click()

    # Verificar que la señal fue emitida
    self.assertEqual(result, ["Retroceso"])
```

## Resumen de las Funcionalidades Probadas

La prueba unitaria para el **botón de retroceso** (back\_button) valida las siguientes funcionalidades:

### **Emisión de la Señal back\_pressed:**

- Se prueba que al hacer clic en el botón de retroceso, se emite correctamente la señal personalizada **back\_pressed**. Esta señal es clave para indicar que el usuario ha solicitado realizar una acción de retroceso.

#### Interacción con el Botón de Retroceso:

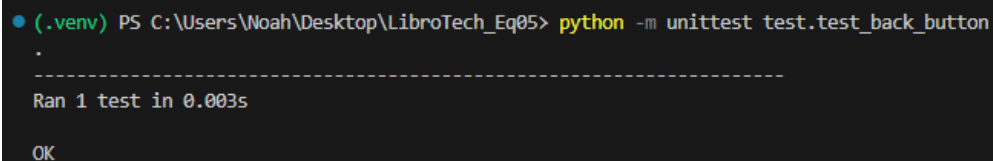
- Se verifica que el clic en el botón de retroceso (representado por el **QPushButton**) sea manejado correctamente y que la señal asociada sea disparada de manera efectiva.

#### Captura de la Señal con un 'Spy':

- Se utiliza una función **espía** (un "spy") para interceptar la señal emitida, asegurando que la acción del clic en el botón realmente resulte en la emisión de la señal esperada.

#### Comprobación de la Funcionalidad de Retroceso:

- La prueba confirma que la señal **back\_pressed** se emite y que se captura correctamente, lo que asegura que la funcionalidad de retroceso en la aplicación está implementada de manera adecuada.



```
(.venv) PS C:\Users\Noah\Desktop\LibroTech_Eq05> python -m unittest test.test_back_button
.
-----
Ran 1 test in 0.003s

OK
```

## 6.- Despliegue de la Aplicación

### 6.1.- Entorno virtual

Comenzaremos hablando sobre el entorno virtual de una aplicación. El entorno virtual permite **aislar las dependencias** de un proyecto en Python, evitando conflictos entre diferentes proyectos que puedan requerir distintas versiones de bibliotecas.

#### 6.1.1.- Cómo crear el entorno virtual en el proyecto.

Para crear un entorno virtual, primero debemos abrir una terminal y navegar al directorio del proyecto. Luego, ejecutamos el siguiente comando:

➤ `python -m venv nombre_entorno`

Reemplazamos **nombre\_entorno** por el nombre que deseamos dar al entorno virtual. Esto creará una carpeta en el directorio del proyecto que contendrá el entorno virtual.



### 6.1.2.- Cómo activar el entorno virtual.

Para activar el entorno virtual, utilizamos los siguientes comandos ubicados en el **directorio raíz** del proyecto y dependiendo del sistema operativo:

- **Windows:** `.\nombre_entorno\Scripts\activate`
- **Linux/Mac:** `source nombre_entorno/bin/activate`

Una vez activado, deberías ver el nombre del entorno en el **prompt** de la terminal, indicando que el entorno virtual está activo.

### 6.1.3.- ¿Qué es el archivo requirements.txt y qué contiene?

El archivo “**requirements.txt**” es un archivo que lista todas las **dependencias** (bibliotecas y sus versiones) necesarias para que la aplicación funcione correctamente. Este archivo permite a otros desarrolladores o entornos de producción instalar fácilmente todas las bibliotecas requeridas.

En nuestro caso, el archivo “**requirements**” contiene las siguientes dependencias:

```
Componentes_EQ05 @ file:./components/componentes_eq05-1.0.tar.gz
psycopg2==2.9.10
psycopg==3.2.1
psycopg-binary==3.2.1
psycopg-pool==3.2.2
PySide6==6.7.2
PySide6_Addons==6.7.2
PySide6_Essentials==6.7.2
setuptools==72.1.0
shiboken6==6.7.2
typing_extensions==4.12.2
tzdata==2024.1
wheel==0.44.0
```

Descripción de las dependencias:

- **psycopg2:** Un adaptador de base de datos PostgreSQL para Python. Es ampliamente utilizado para conectar y trabajar con bases de datos PostgreSQL.
- **Componentes\_EQ05:** Paquete con los componentes personalizados importados.

- **psycopg:** La versión moderna del adaptador de PostgreSQL, que incluye características avanzadas.
- **psycopg-binary:** Una distribución precompilada de psycopg que facilita su instalación sin necesidad de compilación.
- **psycopg-pool:** Proporciona un pool de conexiones para manejar múltiples conexiones a la base de datos de manera eficiente.
- **PySide6:** Es una biblioteca que permite crear interfaces gráficas de usuario (GUI) en Python usando Qt 6.
- **PySide6\_Addons:** Complementos adicionales que extienden la funcionalidad de PySide6.
- **PySide6\_Essentials:** Un conjunto de módulos esenciales de PySide6 que se utilizan para desarrollar aplicaciones con Qt.
- **setuptools:** Una biblioteca que facilita la creación y distribución de paquetes en Python.
- **shiboken6:** Herramienta utilizada para crear bindings entre C++ y Python, fundamental para PySide6.
- **typing\_extensions:** Proporciona funcionalidades de tipado que no están disponibles en todas las versiones de Python.
- **tzdata:** Base de datos de información de zonas horarias, útil para manejar fechas y horas correctamente.
- **wheel:** Un formato de paquete para Python que permite la instalación más rápida de bibliotecas.

#### 6.1.4.- Cómo instalar las dependencias desde el archivo requirements.txt

Para instalar las dependencias listadas en el archivo requirements.txt, aseguramos que el entorno virtual esté **activo** y ejecutamos el siguiente comando:

- `pip install -r requirements.txt`

Este comando instalará automáticamente todas las bibliotecas especificadas en el archivo.

## 6.2.- Cómo convertir un archivo ui en un py.

Para convertir un archivo de diseño de interfaz de usuario (.ui) a un archivo Python (.py), utilizamos la herramienta **pyside6-uic**, que viene con la instalación de PySide6.

El comando que utilizamos es el siguiente:

➤ `pyside6-uic archivo.ui -o archivo.py`

Donde **archivo.ui** es el nombre del archivo de diseño que queremos convertir, y **archivo.py** es el nombre del archivo Python que se generará. Este comando generará el **archivo.py** que contendrá la lógica necesaria para cargar la interfaz definida en **interfaz.ui**.

## 6.3.- Docker y BBDD PostgreSQL.

Antes de explicar cómo montar la base de datos y demás, deberemos tener el servicio de Docker activo, para ello, si estamos en **Windows** simplemente abriremos la aplicación de Docker Desktop.

### 6.3.1.- ¿Qué es Docker Compose y para que se utiliza?

Docker Compose es una herramienta que se utiliza para **definir y ejecutar aplicaciones** que constan de múltiples contenedores. Con Docker Compose, puedes especificar la configuración de los contenedores, redes y volúmenes necesarios para tu aplicación en un archivo YAML (**docker-compose.yml**). Este archivo contiene todos los servicios que tu aplicación requiere, así como sus configuraciones, dependencias y cómo deben interactuar entre sí.

Docker Compose se utiliza para **simplicar la orquestación** y gestión de aplicaciones compuestas por múltiples contenedores. Su principal ventaja es la **facilidad de uso**, ya que permite iniciar todos los servicios definidos en un archivo “**docker-compose.yml**” con un solo comando (**docker-compose up**).

Además, también permite gestionar las dependencias entre contenedores, asegurando que se inicien en el orden correcto, lo que es esencial para aplicaciones que dependen de otros servicios, como bases de datos.

➤ Docker. (n.d.). *Docker documentation*. Docker. <https://docs.docker.com/>

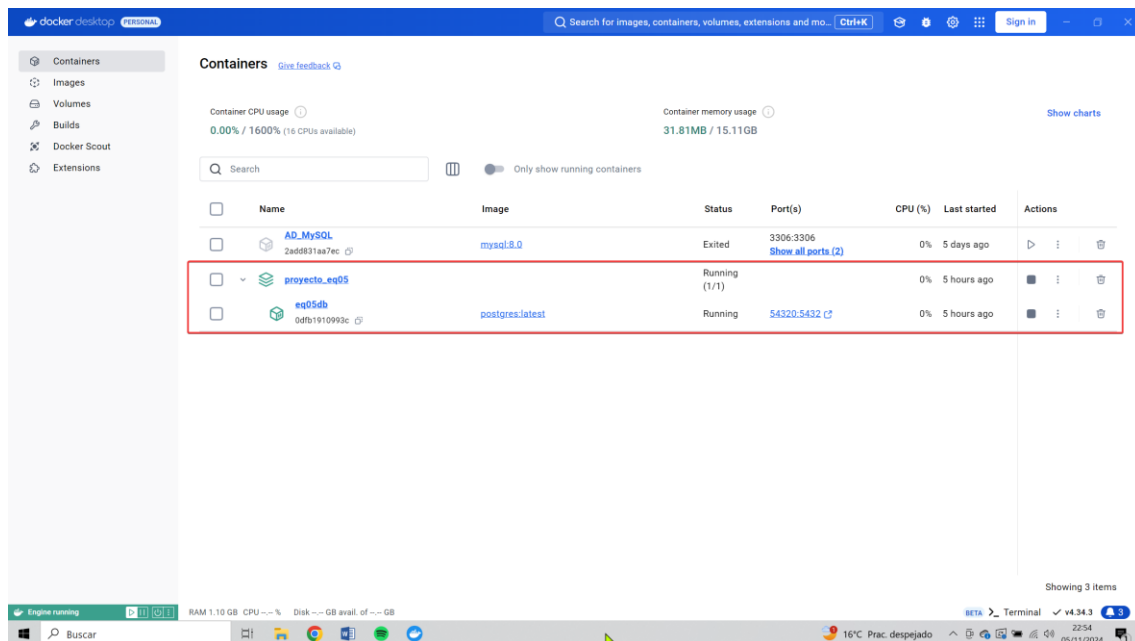
- Microsoft. (n.d.). Receta de Docker Compose para servicios de IA en contenedores de Azure. Microsoft Learn. <https://learn.microsoft.com/es-es/azure/ai-services/containers/docker-compose-recipe>

### 6.3.2.- Levantar el Contenedor

Una vez abierto Docker Desktop, procederemos a lanzar el siguiente comando en una terminal:

- `docker-compose up --build`

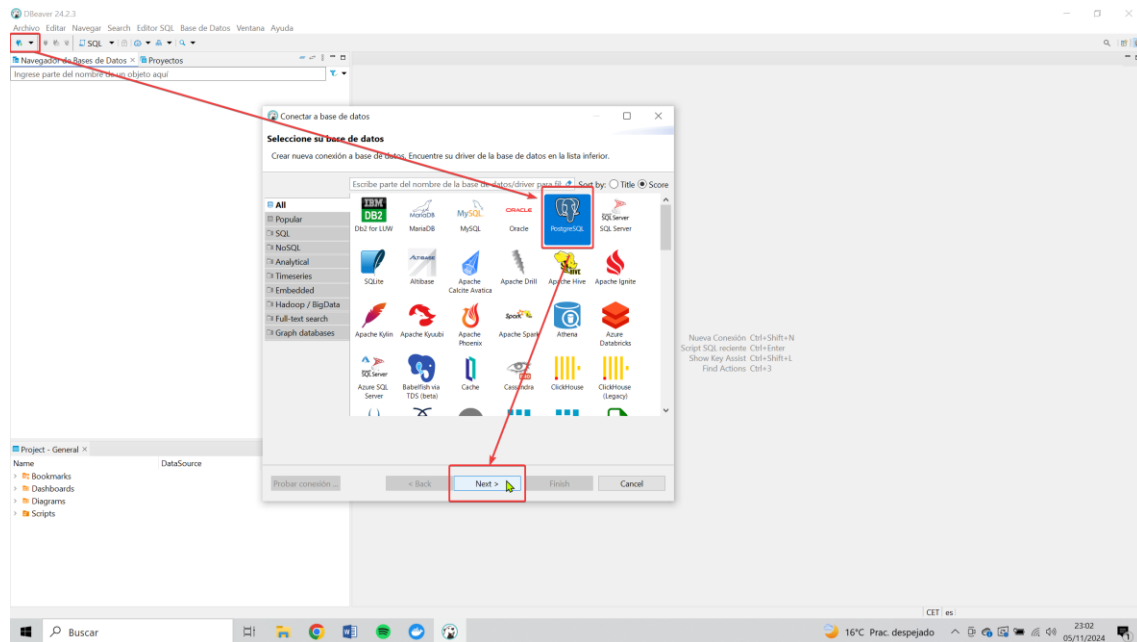
El comando recogerá todos los datos que hayamos introducido en el “docker-compose.yml” y nos creará un contenedor en Docker Desktop. Si lo hemos hecho correctamente, podemos ver que en la lista de contenedores de Docker saldrá esto:



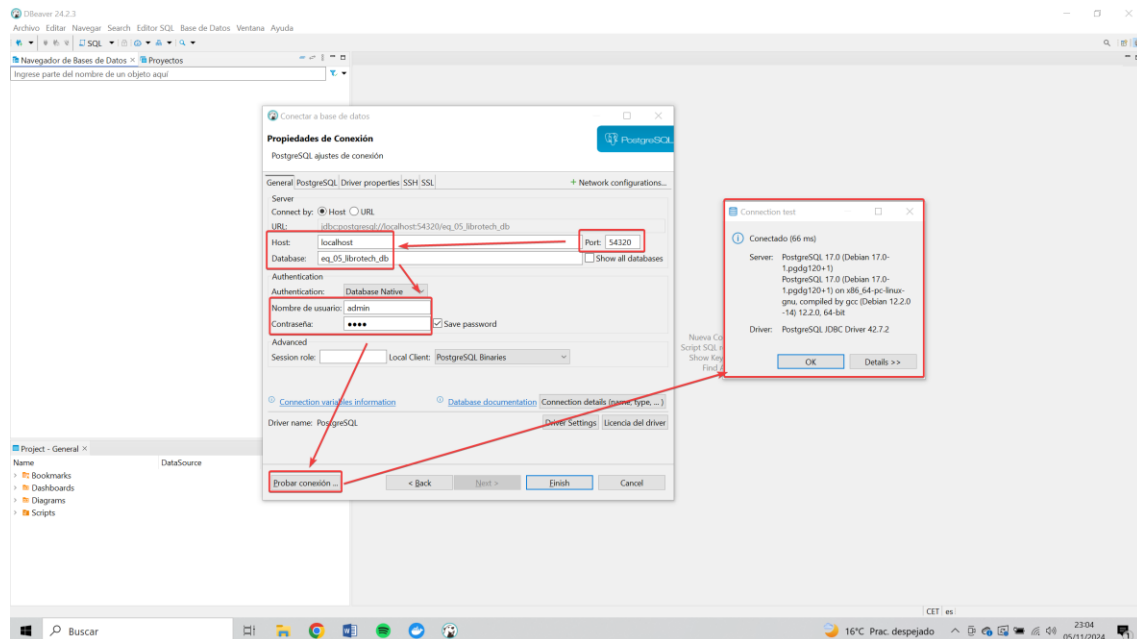
### 6.4.- Conectarnos a la BBDD desde DBeaver.

Para gestionar la base de datos PostgreSQL que hemos desplegado en un contenedor de Docker, utilizaremos DBeaver, una herramienta de administración de bases de datos que facilita la visualización y manipulación de datos. Con DBeaver, podremos establecer una conexión directa con nuestra base de datos en contenedor, verificar su estructura, ejecutar consultas SQL y realizar otras tareas de administración sin necesidad de trabajar directamente en la terminal.

Para conectarnos a la base de datos, una vez abramos el programa pulsaremos en el icono del enchufe arriba a la izquierda, luego seleccionaremos el tipo de base de datos y le daremos a next:



A continuación, nos aparecerá una pestaña donde agregaremos los datos de nuestra base de datos que hicimos en el “docker\_compose.yml”. Para comprobar que están bien, pulsamos sobre “Probar conexión” y si todo va bien, podremos darle a “Finish” y continuar.



Para poder ver los datos de nuestra BBDD, abriremos un script y escribiremos “**select \* from usuarios**” para poder ver todos nuestros usuarios.

