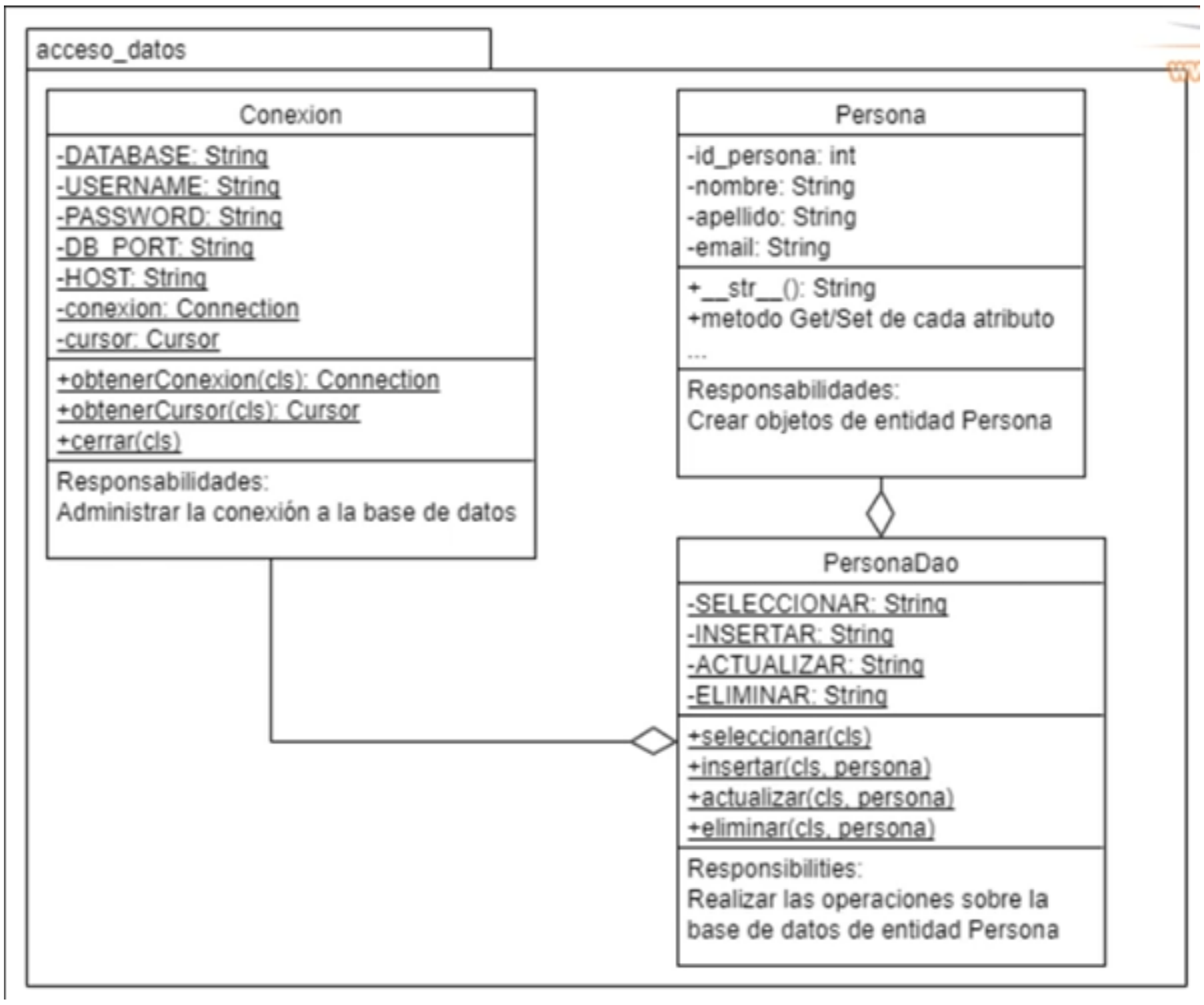
Creación de una capa de datos

Durante esta lección vamos a desarrollar un proyecto que nos permita crear objetos de entidad (registros de la Base de Datos) de tipo `Persona`, realizar `querys` con los datos de estos objetos a través de la clase `PersonaDAO` (`DAO = Data Access Object`) y finalmente manejar la conexión a la Base de Datos con una clase `Conexion`. De este modo, estaremos utilizando conceptos de POO y Bases de Datos para administrar los registros de la Base de Datos con la que hemos estado trabajando. Utilizaremos el siguiente UML para desarrollar la aplicación:



Manejo de `Logging` con Python

El módulo `logging` en Python proporciona una forma flexible y poderosa de registrar información durante la ejecución de un programa. Permite controlar el nivel de detalle de los mensajes de registro, redirigir los mensajes a diferentes destinos (consola, archivos, etc.), y establecer un formato uniforme para los registros.

La forma correcta de hacer `Logging` de acuerdo a la documentación oficial de Python se puede revisar a continuación:

> https://docs.python.org/es/3/howto/logging.html

Este código muestra cómo configurar y utilizar el módulo de registro (`logging`) en Python para registrar mensajes de diferentes niveles en un archivo y en la consola.

```
import logging as log
```

```
# Configuración básica del logger
log.basicConfig(level=log.DEBUG,  # Establece el nivel mínimo de mensajes que se regis
trarán
                format='%(asctime)s: %(levelname)s [%(filename)s: %(lineno)s] %(messag
e)s',  # Formato de los mensajes de registro
                datefmt='%I:%M:%S %p',  # Formato de la fecha y hora
                handlers=[
                                        # Manejador para registrar en un archivo
                    log.FileHandler('C:\\CURSOS\\Python\\BD\\Leccion03 - Capa de Datos
\\capa_datos.log'),
                    log.StreamHandler()  # Manejador para registrar en la consola
                ])

if __name__ == '__main__':
    # Ejemplos de registro de diferentes niveles
    log.debug('Mensaje a nivel debug')  # Mensaje de depuración
    log.info('Mensaje a nivel info')  # Mensaje informativo
    log.warning('Mensaje a nivel warning')  # Advertencia
    log.error('Mensaje a nivel error')  # Error
    log.critical('Mensaje a nivel crítico')  # Error crítico
```

Explicación:

- Se importa el módulo `logging` como `log`.
- Se configura el registro básico mediante `basicConfig()`. Se establece el nivel mínimo de mensajes a `DEBUG`, lo que significa que todos los mensajes de nivel `DEBUG` o superior serán registrados. El formato de los mensajes de registro se define utilizando una cadena de formato, que incluye información como la marca de tiempo, el nivel de registro, el nombre del archivo, el número de línea y el mensaje en sí. También se especifica el formato de la fecha y hora.
- Se definen dos manejadores de registro: uno para registrar en un archivo llamado `capa_datos.log` en la ruta `C:\\CURSOS\\Python\\BD\\Leccion03 - Capa de Datos\\` y otro para registrar en la consola.
- Se comprueba si el script se está ejecutando como el programa principal (`if __name__ == '__main__':`). Esto es comúnmente utilizado para asegurarse de que el código dentro de este bloque solo se ejecute cuando se ejecuta el script directamente, no cuando se importa como un módulo.
- Se registran mensajes de diferentes niveles (`DEBUG`, `INFO`, `WARNING`, `ERROR`, y `CRITICAL`) utilizando las funciones de registro proporcionadas por el módulo `logging`. Cada función de registro corresponde a un nivel de severidad diferente. Los mensajes se enviarán a los manejadores configurados según el nivel de registro especificado en la configuración básica.

Las clases para la creación de la capa de datos se presentan a continuación:

Clase `Persona`

```
from logger_base import *

class Persona:
    def __init__(self, id_persona = None, nombre = None, apellido = None, mail = None):
        self._id_persona = id_persona
        self._nombre = nombre
        self._apellido = apellido
        self._mail = mail

    def __str__(self):
        return  f'''
                ID: {self._id_persona}
```

```
                Nombre: {self._nombre}
                Apellido = {self._apellido}
                Correo: {self._mail}
                '''

    @property
    def id_persona(self):
        return self._id_persona

    @id_persona.setter
    def id_persona(self, id_persona):
        self._id_persona = id_persona

    @property
    def nombre(self):
        return self._nombre

    @nombre.setter
    def nombre(self, nombre):
        self._nombre

    @property
    def apellido(self):
        return self._apellido

    @apellido.setter
    def apellido(self, apellido):
        self._apellido

    @property
    def mail(self):
        return self._mail

    @mail.setter
    def mail(self, mail):
        self._mail

if __name__ == '__main__':
    persona1 = Persona(1, 'Juan', 'Pérez', 'jperez@mail.com')
    log.debug(persona1)

    # Simular un INSERT INTO
    persona1 = Persona(nombre='Juan', apellido='Pérez', mail='jperez@mail.com')
    log.debug(persona1)

    #Simular un DELETE
    persona1 = Persona(id_persona=1)
    log.debug(persona1)
```

Clase `PersonaDAO`

```
from logger_base import *
from Persona import *
from Conexion import *

class PersonaDAO:
    _SELECCIONAR = 'SELECT * FROM persona ORDER BY id_persona'
    _INSERTAR = 'INSERT INTO persona(nombre, apellido, mail) VALUES(%s, %s, %s)'
```

```
    _ACTUALIZAR = 'UPDATE persona SET nombre = %s, apellido = %s, mail = %s WHERE id_pers
    _BORRAR = 'DELETE FROM persona WHERE id_persona = %s'

    @classmethod
    def seleccionar(cls):
        with Conexion.obtener_conexion() as conexion:
            with conexion.cursor() as cursor:
                cursor.execute(cls._SELECCIONAR)
                registros = cursor.fetchall()
                personas = []
                for registro in registros:
                    persona = Persona(registro[0], registro[1], registro[2], registro[3]
                    personas.append(persona)
                return personas

    @classmethod
    def insertar(cls, persona):
        with Conexion.obtener_conexion():
            with Conexion.obtener_cursor() as cursor:
                valores = (persona.nombre, persona.apellido, persona.mail)
                cursor.execute(cls._INSERTAR, valores)
                log.debug(f'Registro insertado en la base de datos: {persona}')
                return cursor.rowcount

    @classmethod
    def actualizar(cls, persona):
        with Conexion.obtener_conexion():
            with Conexion.obtener_cursor() as cursor:
                valores = (persona.nombre, persona.apellido, persona.mail, persona.id_per
                cursor.execute(cls._ACTUALIZAR, valores)
                log.debug(f'Registro actualizado: {persona}')
                return cursor.rowcount

    @classmethod
    def borrar(cls, persona):
        with Conexion.obtener_conexion():
            with Conexion.obtener_cursor() as cursor:
                valores = (persona.id_persona, )
                cursor.execute(cls._BORRAR, valores)
                log.debug(f'Registro eliminado: {persona}')
                return cursor.rowcount
```

```
if __name__ == '__main__':
    """ # Insertar un registro
    persona1 = Persona(nombre = 'Pedro', apellido = 'Nájera', mail = 'pnajera@mail.com')
    personas_insertadas = PersonaDAO.insertar(persona1)
    log.debug(f'Registro insertado en la base de datos: {personas_insertadas}')
    """

    """ # Actualizar un registro
    persona1 = Persona(1, 'Carlos', 'Juárez', 'cjuarez@mail.com')
    personas_actualizadas = PersonaDAO.actualizar(persona1)
    log.debug(f'Registros actualizados: {personas_actualizadas}')
    """

    # Eliminar un registro
    persona1 = Persona(id_persona = 13)
    personas_eliminadas = PersonaDAO.borrar(persona1)
    log.debug(f'Registros eliminados: {personas_eliminadas}')
```

```
    # Seleccionar objetos
    personas = PersonaDAO.seleccionar()
    for persona in personas:
        log.debug(persona)
```

Clase `Conexion`

```
import sys
import psycopg2 as DB
from logger_base import *

class Conexion:
    _DATABASE = 'test_db'
    _USERNAME = 'postgres'
    _PASSWORD = 'admin'
    _DB_PORT = '5432'
    _HOST = '127.0.0.1'
    _conexion = None
    _cursor = None

    @classmethod
    def obtener_conexion(cls):
        if cls._conexion is None:
            try:
                cls._conexion = DB.connect(
                    host = cls._HOST,
                    user = cls._USERNAME,
                    password = cls._PASSWORD,
                    port = cls._DB_PORT,
                    database = cls._DATABASE
                )
                log.debug(f'Conexión exitosa: {cls._conexion}')
                return cls._conexion
            except Exception as e:
                log.error('Ocurrió una excepción al obtener la conexión: {e}')
                sys.exit()
        else:
            return cls._conexion

    @classmethod
    def obtener_cursor(cls):
        if cls._cursor is None:
            try:
                cls._cursor = cls.obtener_conexion().cursor()
                log.debug(f'Cursor abierto correctamente: {cls._cursor}')
                return cls._cursor
            except Exception as e:
                log.error(f'Ocurrió una excepción al obtener el cursor: {e}')
                sys.exit()
        else:
            return cls._cursor


if __name__ == '__main__':
    Conexion.obtener_conexion()
    Conexion.obtener_cursor()
```

Este ejercicio nos ha mostrado cómo utilizar los conceptos de Programación Orientada a Objetos y Bases de Datos en conjunto para el manejo de objetos creados a partir de la clase Persona, utilizándolos como registros de nuestra Base de Datos.

Sin embargo, puede ser implementado de forma más correcta y eficiente utilizando el concepto de `Pool de Conexiones`, el cual veremos a continuación.

`Pool` de conexiones con Python y `postgreSQL`

El "Pool de Conexiones" es una técnica utilizada en programación para gestionar y reutilizar conexiones a recursos externos, como bases de datos, servicios web o servidores de archivos, de una manera eficiente. En lugar de abrir y cerrar conexiones a estos recursos cada vez que se necesita realizar una operación, el pool de conexiones mantiene un conjunto predefinido de conexiones abiertas y disponibles para ser utilizadas cuando sea necesario.

Aquí hay una explicación detallada de cómo funciona un pool de conexiones:

1. **Inicialización del Pool**: En primer lugar, se crea un conjunto inicial de conexiones al recurso externo. Este número puede ser configurado según los requisitos de la aplicación. Por ejemplo, en una aplicación que interactúa con una base de datos, se podría establecer un pool de conexiones con un tamaño inicial de 5 conexiones.
2. **Solicitud de Conexión**: Cuando la aplicación necesita realizar una operación que requiere una conexión al recurso externo, solicita una conexión al pool. En lugar de abrir una nueva conexión cada vez, la aplicación toma una conexión existente del pool, si está disponible.
3. **Utilización de la Conexión**: Una vez que la aplicación obtiene una conexión del pool, realiza la operación necesaria utilizando esa conexión. Esto podría ser una consulta a una base de datos, una solicitud a un servicio web, etc.
4. **Liberación de la Conexión**: Después de que la operación se completa, la aplicación libera la conexión de vuelta al pool en lugar de cerrarla. Esto significa que la conexión no se cierra realmente, sino que se pone de nuevo a disposición para ser utilizada por otra parte de la aplicación.
5. **Administración del Pool**: El pool de conexiones se encarga de administrar las conexiones disponibles, asegurándose de que no se exceda el tamaño máximo del pool y de que las conexiones se mantengan activas y en buen estado. Si una conexión falla o se vuelve inutilizable por algún motivo, el pool puede eliminarla y crear una nueva en su lugar.
6. **Gestión de la Concurrencia**: Los pools de conexiones están diseñados para ser utilizados en entornos concurrentes, donde múltiples partes de la aplicación pueden necesitar acceder al recurso externo al mismo tiempo. El pool gestiona la concurrencia de manera que múltiples solicitudes de conexión simultáneas sean manejadas de manera segura y eficiente.

En resumen, el pool de conexiones ofrece una manera eficiente de gestionar y reutilizar conexiones a recursos externos en aplicaciones que requieren acceso frecuente a esos recursos. Al mantener un conjunto predefinido de conexiones abiertas y disponibles, el pool reduce el tiempo de latencia y el costo de abrir y cerrar conexiones, mejorando así el rendimiento y la escalabilidad de la aplicación.

La forma en la que actualizaremos nuestra capa de datos para que utilice este concepto se muestra a continuación:

Clase `Conexion`

Este código implementa un pool de conexiones utilizando la biblioteca `psycopg2` para interactuar con una base de datos PostgreSQL. A continuación, explicaré cada parte del código:

```
import sys
from psycopg2 import pool
```

```
from logger_base import *
```

- Se importan los módulos necesarios. `sys` se utiliza para interactuar con el sistema operativo, `psycopg2.pool` proporciona las herramientas para crear y administrar el pool de conexiones, y `logger_base` es un módulo personalizado que contiene la configuración de registro.

```
class Conexion:
    _DATABASE = 'test_db'
    _USERNAME = 'postgres'
    _PASSWORD = 'admin'
    _DB_PORT = '5432'
    _HOST = '127.0.0.1'
    _MIN_CONEXIONES = 1
    _MAX_CONEXIONES = 5
    _pool = None
```

- Se define una clase llamada `Conexion`. Dentro de la clase, se definen variables de clase para almacenar la configuración de la conexión y los límites del pool.

```
    @classmethod
    def obtener_pool(cls):
        if cls._pool is None:
            try:
                cls._pool = pool.SimpleConnectionPool(
                    cls._MIN_CONEXIONES,
                    cls._MAX_CONEXIONES,
                    host=cls._HOST,
                    user=cls._USERNAME,
                    password=cls._PASSWORD,
                    port=cls._DB_PORT,
                    database=cls._DATABASE
                )
                log.debug(f'Creación exitosa del pool de conexiones: {cls._pool}')
                return cls._pool
            except Exception as e:
                log.error(f'Ocurrió un error al obtener el pool de conexiones: {e}')
                sys.exit()
        else:
            return cls._pool
```

- `obtener_pool()` es un método de clase que crea y devuelve el pool de conexiones si aún no está creado. Utiliza `psycopg2.pool.SimpleConnectionPool` para crear el pool de conexiones con los parámetros de configuración definidos en la clase.

```
    @classmethod
    def obtener_conexion(cls):
        conexion = cls.obtener_pool().getconn()
        log.debug(f'Conexion obtenida del pool: {conexion}')
        return conexion
```

- `obtener_conexion()` es un método de clase que obtiene una conexión del pool utilizando `getconn()`. Registra en el log la conexión obtenida y la devuelve.

```
    @classmethod
    def liberar_conexion(cls, conexion):
        cls.obtener_pool().putconn(conexion)
        log.debug(f'Conexión regresada al pool de conexiones: {conexion}')
```

- `liberar_conexion()` es un método de clase que libera una conexión y la devuelve al pool utilizando `putconn()`. Registra en el log la conexión liberada.

```
    @classmethod
    def cerrar_conexiones(cls):
        cls.obtener_pool().closeall()
```

- `cerrar_conexiones()` es un método de clase que cierra todas las conexiones del pool utilizando `closeall()`.

```
if __name__ == '__main__':
    conexion1 = Conexion.obtener_conexion()
    Conexion.liberar_conexion(conexion1)

    conexion2 = Conexion.obtener_conexion()
    conexion3 = Conexion.obtener_conexion()
    Conexion.liberar_conexion(conexion3)

    conexion4 = Conexion.obtener_conexion()
    conexion5 = Conexion.obtener_conexion()
    Conexion.liberar_conexion(conexion5)

    conexion6 = Conexion.obtener_conexion()
```

- En el bloque `__main__`, se realizan varias operaciones para demostrar el funcionamiento del pool de conexiones. Se obtienen conexiones, se liberan y se vuelven a obtener. En el último caso, se intenta obtener una conexión después de que todas las conexiones disponibles en el pool han sido utilizadas.

En resumen, este código implementa un pool de conexiones para una base de datos PostgreSQL utilizando `psycopg2`. Proporciona métodos para obtener conexiones del pool, liberar conexiones de vuelta al pool y cerrar todas las conexiones. Además, utiliza un sistema de registro para registrar eventos importantes, como la creación del pool, la obtención y liberación de conexiones.

Clase `CursorDelPool`

Este código define una clase llamada `CursorDelPool`, que implementa un gestor de contexto para trabajar con un cursor de base de datos obtenido de un pool de conexiones. A continuación, explicaré cada parte del código:

```
from logger_base import *
from Conexion import *
```

- Se importan los módulos necesarios. `logger_base` contiene la configuración del registro y `Conexion` es el módulo que contiene la clase `Conexion`, que proporciona métodos para obtener y liberar conexiones de un pool.

```
class CursorDelPool:
    def __init__(self):
        self._conexion = None
        self._cursor = None
```

- Se define la clase `CursorDelPool`. En su inicialización (`__init__`), se definen dos variables de instancia: `_conexion` para almacenar la conexión obtenida del pool y `_cursor` para almacenar el cursor de la conexión.

```
    def __enter__(self):
        log.debug(f'Inicio del método with __enter__')
        self._conexion = Conexion.obtener_conexion()
```

```
        self._cursor = self._conexion.cursor()
        return self._cursor
```

- El método `__enter__` es llamado cuando se entra en el bloque `with` que utiliza una instancia de `CursorDelPool`. Dentro de este método, se obtiene una conexión del pool mediante `Conexion.obtener_conexion()` y se crea un cursor a partir de esa conexión. Luego, se devuelve el cursor para que pueda ser utilizado dentro del bloque `with`.

```
    def __exit__(self, tipo_excepcion, valor_excepcion, detalle_excepcion):
        log.debug(f'Se ejecuta el método __exit__')
        if valor_excepcion:
            self._conexion.rollback()
            log.error(f'Ocurrió una excepción, rollback realizado: {valor_excepcion}
{tipo_excepcion} {detalle_excepcion}')
        else:
            self._conexion.commit()
            log.debug(f'Commit de la transacción')
        self._cursor.close()
        Conexion.liberar_conexion(self._conexion)
```

- El método `__exit__` es llamado cuando se sale del bloque `with`. Aquí se maneja cualquier excepción que ocurra dentro del bloque `with`. Si ocurre una excepción (`valor_excepcion` tiene un valor), se realiza un rollback en la conexión para deshacer cualquier cambio pendiente. Si no hay excepciones, se realiza un commit en la conexión para confirmar los cambios. Luego, se cierra el cursor y se libera la conexión de vuelta al pool mediante `Conexion.liberar_conexion()`.

```
if __name__ == '__main__':
    with CursorDelPool() as cursor:
        log.debug(f'Dentro del bloque with')
        cursor.execute('SELECT * FROM persona')
        log.debug(cursor.fetchall())
```

- En el bloque `__main__`, se utiliza la clase `CursorDelPool` como un gestor de contexto dentro de un bloque `with`. Dentro de este bloque, se ejecutan operaciones utilizando el cursor obtenido. En este ejemplo, se ejecuta una consulta SQL para seleccionar todas las filas de la tabla `persona` y se registran los resultados en el registro de depuración. Al salir del bloque `with`, el método `__exit__` de `CursorDelPool` se ejecuta automáticamente para cerrar el cursor y liberar la conexión de vuelta al pool.

Clase `PersonaDAO`

Este código implementa un objeto de acceso a datos (DAO, por sus siglas en inglés) para realizar operaciones CRUD (crear, leer, actualizar y eliminar) en una tabla de base de datos llamada `persona`. A continuación, explicaré cada parte del código:

```
from logger_base import *
from Persona import *
from Conexion import *
from cursor_del_pool import *
```

- Se importan los módulos necesarios. `logger_base` contiene la configuración del registro, `Persona` contiene la definición de la clase `Persona`, `Conexion` contiene la clase `Conexion` que gestiona el pool de conexiones, y `cursor_del_pool` contiene la clase `CursorDelPool` que proporciona un gestor de contexto para trabajar con un cursor obtenido del pool de conexiones.

```
class PersonaDAO:
    _SELECCIONAR = 'SELECT * FROM persona ORDER BY id_persona'
    _INSERTAR = 'INSERT INTO persona(nombre, apellido, mail) VALUES(%s, %s, %s)'
```

```
    _ACTUALIZAR = 'UPDATE persona SET nombre = %s, apellido = %s, mail = %s WHERE id_p
ersona = %s'
    _BORRAR = 'DELETE FROM persona WHERE id_persona = %s'
```

- Se define la clase `PersonaDAO`. Dentro de la clase, se definen constantes para las consultas SQL utilizadas en las operaciones CRUD.

```
    @classmethod
    def seleccionar(cls):
        with CursorDelPool() as cursor:
            cursor.execute(cls._SELECCIONAR)
            registros = cursor.fetchall()
            personas = []
            for registro in registros:
                persona = Persona(registro[0], registro[1], registro[2], registro[3])
                personas.append(persona)
            return personas
```

- El método `seleccionar` realiza una consulta para seleccionar todos los registros de la tabla `persona`. Utiliza un cursor obtenido del pool de conexiones y ejecuta la consulta. Luego, crea objetos `Persona` a partir de los registros obtenidos y los devuelve como una lista.

```
    @classmethod
    def insertar(cls, persona):
        with CursorDelPool() as cursor:
            valores = (persona.nombre, persona.apellido, persona.mail)
            cursor.execute(cls._INSERTAR, valores)
            log.debug(f'Registro insertado en la base de datos: {persona}')
            return cursor.rowcount
```

- El método `insertar` inserta un nuevo registro en la tabla `persona`. Utiliza un cursor obtenido del pool de conexiones para ejecutar la consulta de inserción con los valores de la instancia de `Persona` proporcionada. Registra el evento en el registro y devuelve el número de filas afectadas.

```
    @classmethod
    def actualizar(cls, persona):
        with CursorDelPool() as cursor:
            valores = (persona.nombre, persona.apellido, persona.mail, persona.id_pers
ona)
            cursor.execute(cls._ACTUALIZAR, valores)
            log.debug(f'Registro actualizado: {persona}')
            return cursor.rowcount
```

- El método `actualizar` actualiza un registro existente en la tabla `persona`. Utiliza un cursor obtenido del pool de conexiones para ejecutar la consulta de actualización con los valores actualizados de la instancia de `Persona` proporcionada. Registra el evento en el registro y devuelve el número de filas afectadas.

```
    @classmethod
    def borrar(cls, persona):
        with CursorDelPool() as cursor:
            valores = (persona.id_persona, )
            cursor.execute(cls._BORRAR, valores)
            log.debug(f'Registro eliminado: {persona}')
            return cursor.rowcount
```

- El método `borrar` elimina un registro de la tabla `persona` basado en el `id_persona` proporcionado. Utiliza un cursor obtenido del pool de conexiones para ejecutar la consulta de eliminación. Registra el evento en el registro y devuelve el número de filas afectadas.

```
if __name__ == '__main__':
    # Insertar un registro
    persona1 = Persona(nombre='Alejandra', apellido='Téllez', mail='atellez@mail.com')
    personas_insertadas = PersonaDAO.insertar(persona1)
    log.debug(f'Registro insertado en la base de datos: {personas_insertadas}')

    # Actualizar un registro
    persona1 = Persona(1, 'Juan', 'Pérez', 'jperez@mail.com')
    personas_actualizadas = PersonaDAO.actualizar(persona1)
    log.debug(f'Registros actualizados: {personas_actualizadas}')

    # Eliminar un registro
    persona1 = Persona(id_persona=16)
    personas_eliminadas = PersonaDAO.borrar(persona1)
    log.debug(f'Registros eliminados: {personas_eliminadas}')

    # Seleccionar objetos
    personas = PersonaDAO.seleccionar()
    for persona in personas:
        log.debug(persona)
```

- En el bloque `__main__`, se realizan varias operaciones de ejemplo para demostrar el uso del `PersonaDAO`. Se inserta un nuevo registro, se actualiza un registro existente, se elimina un registro y se seleccionan todos los registros de la tabla. Los eventos se registran en el registro para su seguimiento y depuración.

Laboratorio: Capa de Datos de Usuarios

Para este ejercicio, se reutilizaron las clases `Conexion`, `LoggerBase` y `CursorDelPool`, cambiando únicamente cuestiones relacionadas con el directorio donde se guarda el log, la tabla de la base de datos `usuario` y el formato de la impresión de texto en consola.

El código de las clases `Usuario`, `UsuarioDAO` y `MenuAppUsuario` se muestra a continuación:

Clase `Usuario`

```
from logger_base import *


class Usuario:
    def __init__(self, id_usuario = None, username = None, password = None):
        self._id_usuario = id_usuario
        self._username = username
        self._password = password

    def __str__(self):
        return f'''
        ID: {self._id_usuario}
        Nombre de usuario: {self._username}
        Contraseña: {self._password}
        '''

    @property
    def id_usuario(self):
        return self._id_usuario
```

```
    @id_usuario.setter
    def id_usuario(self, id_usuario):
        self._id_usuario = id_usuario

    @property
    def username(self):
        return self._username

    @username.setter
    def username(self, username):
        self._username = username

    @property
    def password(self):
        return self._password

    @password.setter
    def password(self, password):
        self._password = password


if __name__ == '__main__':
    usuario1 = Usuario(1, 'eacrespo', '3acr3sp0')
    log.debug(usuario1)

    # Simulación de inserción
    usuario1 = Usuario(username='eacrespo', password='3acr3sp0')
    log.debug(usuario1)
```

Clase `UsuarioDAO`

```
from logger_base import *
from Usuario import *
from Conexion import *
from CursorDelPool import *

class UsuarioDAO:
    _SELECCIONAR = 'SELECT * FROM usuario ORDER BY id_usuario'
    _INSERTAR = 'INSERT INTO usuario(username, password) VALUES(%s, %s)'
    _ACTUALIZAR = 'UPDATE usuario SET username = %s, password = %s WHERE id_usuario = %s
    _BORRAR = 'DELETE FROM usuario WHERE id_usuario = %s'

    @classmethod
    def seleccionar(cls):
        with CursorDelPool() as cursor:
            cursor.execute(cls._SELECCIONAR)
            registros = cursor.fetchall()
            usuarios = []
            for registro in registros:
                usuario = Usuario(registro[0], registro[1], registro[2])
                usuarios.append(usuario)
            return usuarios

    @classmethod
    def insertar(cls, usuario):
        with CursorDelPool() as cursor:
            valores = (usuario.username, usuario.password)
            cursor.execute(cls._INSERTAR, valores)
```

```
            log.debug(f'Registro insertado en la Base de Datos:\n{usuario}')
            return cursor.rowcount

    @classmethod
    def actualizar(cls, usuario):
        with CursorDelPool() as cursor:
            valores = (usuario.username, usuario.password, usuario.id_usuario)
            cursor.execute(cls._ACTUALIZAR, valores)
            log.debug(f'Registro actualizado en la Base de Datos:\n{usuario}')
            return cursor.rowcount

    @classmethod
    def borrar(cls, usuario):
        with CursorDelPool() as cursor:
            valores = (usuario.id_usuario, )
            cursor.execute(cls._BORRAR, valores)
            log.debug(f'Registro eliminado de la Base de Datos:\n{usuario}')
            return cursor.rowcount


if __name__ == '__main__':
    # Inserción de un registro
    usuario1 = Usuario(username='SirRinoceronte', password='r1n0c3r0nt3')
    usuarios_insertados = UsuarioDAO.insertar(usuario1)
    log.debug(f'Registro insertado en la Base de Datos:\n{usuarios_insertados}')

    # Seleccionar objetos
    usuarios = UsuarioDAO.seleccionar()
    for usuario in usuarios:
        log.debug(usuario)
```

Clase `MenuAppUsuario`

```
from UsuarioDAO import *

opcion = None

while opcion != 5:
    try:
        print(f'''
        Bienvenido al sistema de administración de usuarios.
        1. Listar usuarios en sistema.
        2. Agregar un usuario al sistema.
        3. Actualizar registro de usuario.
        4. Eliminar registro de usuario.
        5. Salir.
        ''')
        opcion = int(input('Ingresa una opción: '))
        if opcion == 1:
            lista_usuarios = UsuarioDAO.seleccionar()
            for usuario in lista_usuarios:
                log.info(usuario)
        elif opcion == 2:
            usuario_tmp = input('Ingresa el Nombre del usuario: ')
            password_tmp = input('Ingresa la contraseña del usuario: ')
            usuario = Usuario(username=usuario_tmp, password=password_tmp)
            registro_agregado = UsuarioDAO.insertar(usuario)
            log.info(f'Usuario agregado a la Base de Datos: {registro_agregado}')
        elif opcion == 3:
```

```
            id_usuario_tmp = int(input('Ingrese el ID del registo a actualizar: '))
            usuario_tmp = input('Ingrese el nuevo nombre de usuario: ')
            password_tmp = input('Ingrese la nueva contraseña: ')
            usuario = Usuario(id_usuario_tmp, usuario_tmp, password_tmp)
            registro_actualizado = UsuarioDAO.actualizar(usuario)
            log.info(f'Usuario actualizado en la Base de Datos: {registro_actualizado}')
        elif opcion == 4:
            id_usuario_tmp = int(input('Ingrese el ID del registro a eliminar: '))
            usuario = Usuario(id_usuario=id_usuario_tmp)
            registro_borrado = UsuarioDAO.borrar(usuario)
            log.info(f'Usuario eliminado de la Base de Datos: {registro_borrado}')
    except Exception as e:
        log.error(f'Ocurrió una excepción:\n{e}')
        opcion = None
else:
    log.info('Fin de la ejecución')
```