Patrón Singleton y la Gestión de la Base de Datos (PostgreSQL en Django)

Un Singleton es un patrón creacional que básicamente se asegura de que una clase tenga una sola instancia en todo el sistema y proporciona un punto de acceso global a ella.

Sin embargo, investigando un poco, encontramos que en Django, implementar un Singleton manualmente para la conexión de la BD es una mala práctica y, de hecho, el framework ya maneja esto de una forma mucho más eficiente.

¿Por qué un Singleton simple no funciona?

Una aplicación web como la que estamos construyendo con Django maneja múltiples peticiones de usuarios al mismo tiempo. Si tuviéramos una única instancia de conexión (Singleton) para toda la aplicación, cada petición tendría que esperar a que la anterior termine de usar la conexión. Esto crearía un cuello de botella terrible y el rendimiento sería pésimo.

¿Qué usa Django en su lugar?

Django no usa un Singleton, sino un sistema de gestión de conexiones (Connection Pooling).

Django usa una configuración única en el archivo settings.py que actúa como el punto de acceso global a los parámetros de la base de datos, pero no a la conexión en sí. Para evitar la lentitud de crear una conexión nueva con cada petición, Django administra un "pool de conexiones" usando el parámetro CONN_MAX_AGE, el cual le indica que mantenga las conexiones abiertas para reutilizarlas. De esta forma, cuando llega una petición (como la de un recepcionista registrando un socio), Django toma una conexión disponible del pool, realiza la consulta y finalmente libera la conexión para que otra petición pueda utilizarla.

Patrón de Diseño: Factory Method (Fábrica)

Para el segundo patrón, elegimos el Factory Method (Método de Fábrica). Es un patrón creacional que nos viene bien para uno de nuestros requisitos.

El patrón Factory proporciona una interfaz para crear objetos, pero deja que las subclases decidan qué tipo de objeto concreto crear. Esto nos permite desacoplar nuestro código.

Tratemos de usarlo para generación de recibos (PAY-R5)

Nuestro requisito PAY-R5 dice: “El Sistema PERMITE a la Recepción generar un recibo de pago identificable que puede ser impreso o almacenado digitalmente, facilitando el seguimiento de las transacciones.”

Aquí tenemos un problema, el sistema necesita crear "recibos", pero hay al menos dos tipos diferentes: Recibo Impreso (que podría ser un PDF) y Recibo Digital (que podría ser un JSON para guardar en la BD o enviar a un email).

Si no usamos un patrón, el código en nuestra app apps/pagos/vistas.py (donde se manejaría el C.U. "Registrar pago de membresía") se llenaría de if-else para decidir qué tipo de recibo crear. Eso es difícil de mantener.

Solución con Factory Method:

Primero, definimos cómo se ve un recibo en general. Creamos una clase base Recibo.

Python

```
class Recibo:
    def __init__(self, datos_pago):
        self.datos_pago = datos_pago

    def generar(self):
        raise NotImplementedError("La subclase debe
implementar este método.")
```

Segundo, Creamos una clase para cada tipo de recibo que necesitamos.

Python

```
class ReciboPDF(Recibo):
       def generar(self):
        print(f"Generando PDF para el pago
{self.datos_pago['id']}...")
```

```
        return
f"/media/recibos/pago_{self.datos_pago['id']}.pdf"

class ReciboDigital(Recibo):
    def generar(self):
        print(f"Generando JSON para el pago
{self.datos_pago['id']}...")
        return {
            "id_pago": self.datos_pago['id'],
            "monto": self.datos_pago['monto'],
            "socio": self.datos_pago['socio_nombre'],
            "status": "completado"
        }
```

Tercero, Creamos una clase "fábrica" que sabe cómo construir cada recibo.

Python

```
class ReciboFactory:

    @staticmethod
    def crear_recibo(tipo, datos_pago):
        if tipo == 'impreso':
            return ReciboPDF(datos_pago)
        elif tipo == 'digital':
            return ReciboDigital(datos_pago)
        else:
            raise ValueError(f"Tipo de recibo '{tipo}' no es
válido.")
```

Cuarto, Ahora, en nuestra vista apps/pagos/views.py, el código queda súper limpio. La vista no sabe cómo se crea un PDF, solo le pide a la fábrica que lo haga.

Python

```python
# En apps/pagos/views.py

from .recibos import ReciboFactory

def registrar_pago_view(request, pago_id):
    pago = Pago.objects.get(id=pago_id)
    datos_recibo = {
        "id": pago.id,
        "monto": pago.monto,
        "socio_nombre": pago.socio.nombre_completo,
    }

    recibo_pdf = ReciboFactory.crear_recibo('impreso',
datos_recibo)
    recibo_digital = ReciboFactory.crear_recibo('digital',
datos_recibo)

    url_pdf = recibo_pdf.generar()
    json_recibo = recibo_digital.generar()

    return HttpResponse("Pago registrado y recibos
generados.")
```

Ventaja, Si el día de mañana el cliente nos pide "ahora también quiero recibos por Email", solo tenemos que crear la clase Recibo Email y agregar un elif tipo == 'email' en la fábrica. No tendríamos que tocar nada en la lógica de la vista registrar_pago_view. Esto hace el sistema mucho más fácil de mantener y escalar.