1. Introducción

Dentro del desarrollo de software, los **patrones de diseño** son soluciones comprobadas que permiten resolver problemas recurrentes en la arquitectura y organización del código.
Su uso mejora la **mantenibilidad, escalabilidad y reutilización** del sistema, además de garantizar una estructura coherente entre los distintos módulos del proyecto:

- El patrón **Singleton**, implementado directamente sobre la gestión de base de datos
- El patrón **Factory Method**, documentado como propuesta de extensión para la creación de objetos del dominio (por ejemplo, usuarios o reservas).

El propósito principal es demostrar cómo estos patrones pueden **integrarse naturalmente**, optimizando tanto el rendimiento como la claridad arquitectónica del proyecto **BuildingAppProject**.

2. Contexto del Proyecto

El proyecto BuildingAppProject tiene como objetivo gestionar la información de un conjunto residencial, incluyendo **usuarios, apartamentos, reservas, asambleas, zonas comunes** y más.
La estructura del sistema se basa en el patrón **Modelo-Vista-Controlador (MVC)** adaptado a Django (Modelo-Vista-Template), en donde las vistas (views.py) controlan la interacción con los modelos (models.py) y las plantillas HTML (templates/).

En este contexto, la aplicación core define el modelo Usuario, y las vistas permiten listar los usuarios registrados en la base de datos, mediante la función:

```
def listar_usuarios(request):
    todos_los_usuarios = Usuario.objects.all()
    contexto = {'usuarios': todos_los_usuarios}
    return render(request, 'core/listar_usuarios.html', contexto)
```

Esta función depende del **ORM de Django** para realizar consultas SQL automáticas. Sin embargo, cada invocación podría llegar a generar múltiples instancias o conexiones a base de datos si no se controla correctamente la sesión o el acceso concurrente.
Aquí es donde entra la **utilidad del patrón Singleton**, garantizando que solo

exista **una instancia controlada** de conexión o acceso a datos durante la ejecución del proyecto.

3. Patrón Singleton: Justificación y Aplicación

3.1. Descripción del patrón

El patrón **Singleton** garantiza que **solo exista una instancia única de una clase** en todo el sistema, y que esta sea accesible de forma global.
Se utiliza comúnmente para manejar recursos compartidos, como conexiones a base de datos, registros de configuración o controladores de logs.

3.2. Por qué usar Singleton en Django

Aunque Django ya implementa de forma interna un sistema de conexión persistente con la base de datos (a través de su ORM y el archivo settings.py), en algunos contextos puede ser necesario **reforzar este control de acceso único**, especialmente cuando se realizan operaciones específicas o consultas repetitivas fuera del ORM (por ejemplo, para auditorías o reportes).

En este proyecto, la vista views.py puede beneficiarse de un **gestor Singleton** que controle el acceso a las operaciones sobre la base de datos, evitando la creación innecesaria de múltiples objetos de consulta.

4. Implementación del Singleton en views.py

La siguiente adaptación del archivo core/views.py aplica el patrón Singleton **sin crear nuevos archivos**, utilizando únicamente el código existente:

```
from django.shortcuts import render
from django.http import HttpResponse
from .models import Usuario

# Clase Singleton para manejar acceso centralizado
class UsuarioManager:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(UsuarioManager, cls).__new__(cls)
        return cls._instance

    def listar_todos(self):
        return Usuario.objects.all()
```

```
# Vista que utiliza el Singleton
def listar_usuarios(request):
    manager = UsuarioManager()
    usuarios = manager.listar_todos()
    contexto = {'usuarios': usuarios}
    return render(request, 'core/listar_usuarios.html', contexto)

def hola(request):
    return HttpResponse("Hola Django, la app core está funcionando ")
```

4.1. Explicación paso a paso

1. **Definición de la clase UsuarioManager:**
   Centraliza las operaciones sobre el modelo Usuario.
   Esta clase se instancia solo una vez gracias al método especial __new__().
2. **Uso del atributo _instance:**
   Permite almacenar una referencia única al objeto creado. Si la clase ya fue instanciada, simplemente devuelve la misma instancia.
3. **Método listar_todos:**
   Realiza la consulta ORM Usuario.objects.all() encapsulada dentro del Singleton.
4. **En la vista listar_usuarios:**
   Se obtiene la instancia del Singleton (manager = UsuarioManager()), garantizando que todas las consultas al modelo Usuario compartan la misma referencia administradora.
5. **Ventajas obtenidas:**
   - Control único de acceso a los datos del modelo.
   - Reducción de sobrecarga al evitar múltiples instancias.
   - Facilita la extensión futura (por ejemplo, para logging, caché o estadísticas de acceso).

5. Singleton implícito en settings.py

Además del Singleton implementado en las vistas, **Django ya utiliza internamente el patrón Singleton en su sistema de configuración**, particularmente en el archivo settings.py.

Cada vez que se ejecuta el proyecto, Django carga el módulo de configuración y lo mantiene en memoria como una **instancia única global**, accesible mediante:

```
from django.conf import settings
print(settings.DATABASES)
```

Esto demuestra que el propio framework garantiza una única instancia de configuración (Singleton natural).
Por tanto, al aplicar el patrón en las vistas, se refuerza esta idea: tanto la **configuración general** como el **acceso a datos** se gestionan de manera controlada y centralizada.

6. Segundo patrón: Factory Method

6.1. Descripción del patrón

El patrón **Factory Method** propone una forma de crear objetos sin especificar la clase exacta del objeto que se va a crear.
Este patrón delega la responsabilidad de instanciación a subclases o métodos especializados, mejorando la flexibilidad y la extensibilidad del sistema.

6.2. Escenario de aplicación en BuildingAppProject

En este proyecto, podría implementarse para **crear diferentes tipos de usuarios** según su rol dentro del conjunto residencial (Admin, Propietario, Residente).
Actualmente, el modelo Usuario maneja todos los tipos dentro de un único modelo con un campo rol. Sin embargo, mediante el uso del patrón Factory, podríamos modularizar la creación:

```
class UsuarioFactory:
    @staticmethod
    def crear_usuario(rol, nombre, correo, contrasena):
        if rol == 'Admin':
            return Usuario(nombre=nombre, correo=correo, contrasena=contrasena,
rol='Admin')
        elif rol == 'Propietario':
            return Usuario(nombre=nombre, correo=correo, contrasena=contrasena,
rol='Propietario')
        elif rol == 'Residente':
            return Usuario(nombre=nombre, correo=correo, contrasena=contrasena,
rol='Residente')
```

6.3. Beneficios

- Facilita la **extensión de tipos de usuarios** sin modificar la lógica principal.

- Aísla la lógica de creación, cumpliendo con el principio **Open/Closed** (abierto a extensión, cerrado a modificación).
- Permite mantener una **estructura uniforme de instanciación** de objetos dentro del sistema.