¿Qué es Singleton?

Singleton es un patrón de diseño creacional que permite asegurarse de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Su propósito principal es controlar el acceso compartido a recursos o configuraciones comunes, evitando que existan múltiples objetos que puedan causar inconsistencias o uso innecesario de memoria.

¿Cuándo usar Singleton?

Se utiliza Singleton cuando se necesita una sola instancia de una clase que coordine acciones o maneje un recurso global en el sistema. Algunos casos típicos son:

- Cuando se necesita una única conexión a la base de datos durante toda la aplicación.
- Para gestionar configuraciones globales o parámetros compartidos entre módulos.
- Cuando se requiere un punto central de registro de logs, eventos o errores.
- Para coordinar el acceso a recursos compartidos (por ejemplo, un gestor de colas o un servicio de notificaciones).
- En casos donde crear múltiples instancias podría generar conflictos, sobrecarga o datos inconsistentes

Implementación en el proyecto

1. Archivo Principal: `project/singleton.py`

Propósito: Implementación central del patrón Singleton para gestión de configuración global.

Funcionalidad:

- Garantiza una única instancia en memoria mediante el método `__new__`
- Almacena configuración de aplicación en diccionario interno
- Provee interfaz unificada para acceso y modificación de configuración
- Sirve como punto de acceso global para parámetros del sistema

Justificación: Centraliza la gestión de configuración evitando duplicación y garantizando consistencia en toda la aplicación.

2. Archivo de Especialización: `grupos/singletons.py`

Propósito: Implementación de Singleton especializado para gestión de cache de grupos.

Funcionalidad:

- Demuestra la escalabilidad del patrón Singleton
- Maneja operaciones específicas de cache para el módulo de grupos
- Implementa lógica de negocio especializada manteniendo la restricción de instancia única

Justificación: Muestra cómo el patrón puede extenderse para diferentes dominios dentro de la misma aplicación.

3. Archivo de Verificación: `grupos/tests.py`

Propósito: Validación automatizada del correcto funcionamiento del patrón Singleton.

Funcionalidad:

- Verifica que siempre se retorne la misma instancia
- Valida el correcto comportamiento de los métodos de configuración
- Garantiza la integridad del patrón mediante pruebas unitarias

Justificación: Asegura la confiabilidad y mantenibilidad del código mediante verificación automatizada.

4. Archivo de Integración: `grupos/views.py`

Propósito: Demostración práctica de uso del Singleton en el contexto web.

Funcionalidad:

- Integra el Singleton con el framework Django
- Utiliza la configuración centralizada en vistas y endpoints
- Ejemplifica el beneficio del acceso unificado a configuración

Justificación: Proporciona casos de uso reales que demuestran el valor del patrón en producción.

5. Archivo de Configuración: `project/urls.py`

Propósito: Habilitación de endpoints para acceder a funcionalidades del Singleton.

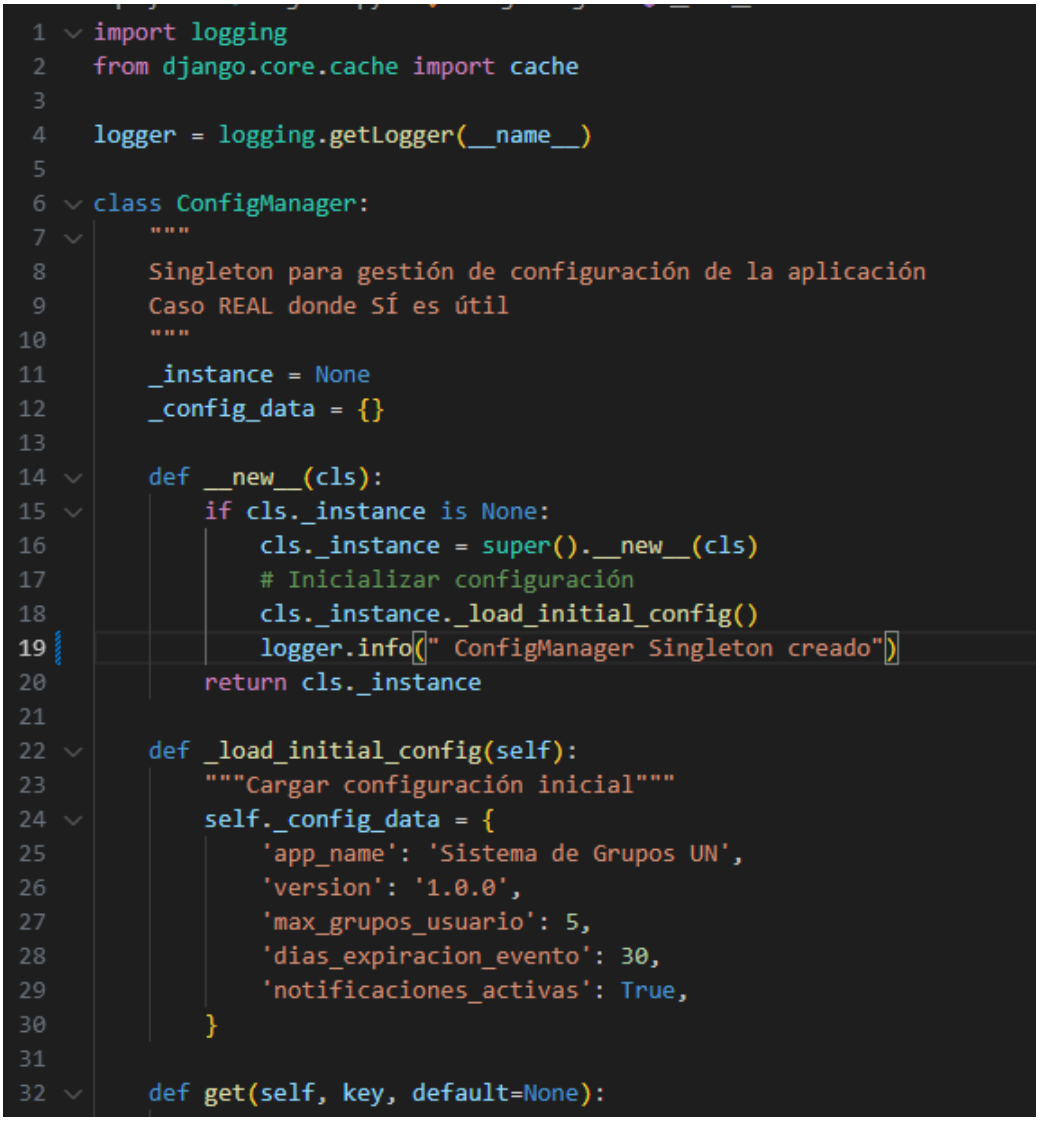
Funcionalidad:

- Configura rutas HTTP para vistas que utilizan el Singleton
- Expone la funcionalidad mediante API web
- Integra el patrón con la capa de presentación

Justificación: Facilita el acceso y prueba de la implementación mediante interfaz web.

Archivos Agregados/Modificados:

1. project/singleton.py - Singleton Principal



```python
import logging
from django.core.cache import cache

logger = logging.getLogger(__name__)

class ConfigManager:
    """
    Singleton para gestión de configuración de la aplicación
    Caso REAL donde SÍ es útil
    """
    _instance = None
    _config_data = {}

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            # Inicializar configuración
            cls._instance._load_initial_config()
            logger.info(" ConfigManager Singleton creado")
        return cls._instance

    def _load_initial_config(self):
        """Cargar configuración inicial"""
        self._config_data = {
            'app_name': 'Sistema de Grupos UN',
            'version': '1.0.0',
            'max_grupos_usuario': 5,
            'dias_expiracion_evento': 30,
            'notificaciones_activas': True,
        }

    def get(self, key, default=None):
```

2. grupos/singletons.py - Singleton Específico para Grupos

```python
from django.core.cache import cache
import logging

logger = logging.getLogger(__name__)

class GrupoCacheManager:
    """
    Singleton para cache específico de grupos
    """
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance._cache_prefix = "grupo_"
            logger.info(" GrupoCacheManager Singleton creado")
        return cls._instance

    def get_grupo(self, grupo_id):
        """Obtener grupo desde cache o BD"""
        cache_key = f"{self._cache_prefix}{grupo_id}"
        grupo_data = cache.get(cache_key)

        if grupo_data is None:
            # Simular carga desde BD
            from .models import Grupo
            try:
                grupo = Grupo.objects.get(id_grupo=grupo_id)
                grupo_data = {
                    'id': grupo.id_grupo,
                    'nombre': grupo.nombre_grupo,
```

3. grupos/tests.py - Tests del Singleton

```python
from django.test import TestCase
from .singletons import grupo_cache
from project.singleton import config_manager

class SingletonTest(TestCase):
    def test_singleton_instances(self):
        """Verificar que siempre es la misma instancia"""
        cache1 = grupo_cache
        cache2 = grupo_cache
        self.assertIs(cache1, cache2)  # Mismo objeto en memoria

        config1 = config_manager
        config2 = config_manager
        self.assertIs(config1, config2)
```

4. grupos/views.py - Uso del Singleton en Vistas

```python
def grupo_detail(request, pk=1):
    grupo = get_object_or_404(Grupo, pk=pk)
    return render(request, "grupos/grupo_detail.html", {"grupo": grupo})

class GrupoDetailView(View):
    def get(self, request, grupo_id):
        """Vista que usa el Singleton de cache"""
        # Usar el singleton para obtener el grupo
        grupo_data = grupo_cache.get_grupo(grupo_id)

        if not grupo_data:
            return JsonResponse({'error': 'Grupo no encontrado'}, status=404)

        return JsonResponse({
            'grupo': grupo_data,
            'config': {
                'app_name': config_manager.get('app_name'),
                'max_grupos': config_manager.get('max_grupos_usuario')
            }
        })

class ConfigView(View):
    def get(self, request):
        """Vista para ver/configurar el Singleton"""
        return JsonResponse({
            'configuracion': config_manager.get_all(),
            'estadisticas_cache': grupo_cache.get_estadisticas()
        })

```

5. Archivo de Configuración: project/urls.py

```python
from django.urls import path
from . import views

urlpatterns = [
    path('grupo/<int:grupo_id>/', views.GrupoDetailView.as_view(), name='grupo_detail'),
    path('config/', views.ConfigView.as_view(), name='config'),
]
```

¿Qué es Observer?

Observer es un patrón de diseño de comportamiento que permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

¿Cuándo usar Observer?

- Cuando varios componentes deben reaccionar a cambios en otro componente sin acoplamiento fuerte.
- Para notificaciones (email/push/in-app), actualización de UI ante cambios en modelos, o sistemas publish/subscribe internos.
- Cuando se busca flexibilidad para añadir/retirar observadores en tiempo de ejecución.

En palabras generales se utiliza el patrón Observer cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.

El patrón Observer permite que cualquier objeto que implemente la interfaz suscriptora pueda suscribirse a notificaciones de eventos en objetos notificadores.

En **ÁgoraUN** los administradores de clubes pueden publicar eventos y los estudiantes pueden suscribirse a clubes para recibir avisos de actividad. Actualmente el sistema persiste eventos en la base de datos, pero no existe un mecanismo desacoplado y extensible para notificar a todos los estudiantes suscritos cuando se publica un evento nuevo.

Problemas observados o potenciales en el proyecto:

- Si la lógica de notificación está embebida en el repositorio, el código queda fuertemente acoplado y poco mantenible.

- Envíos masivos síncronos pueden bloquear peticiones y reducir la disponibilidad.

- Añadir nuevos canales de notificación (email, push, in-app) implicaría modificar múltiples partes del código.

- No hay un mecanismo claro para la suscripción / des-suscripción dinámico de receptores.

Requisito funcional asociado: Notificar a estudiantes suscritos cuando se publica un nuevo evento (RF_05 / RF_09 / RF_06 relacionados).

Solución propuesta

Aplicar el patrón Observer al convertir la publicación de eventos en un `Subject` (observable) que notifica a un conjunto de `Observers` (estudiantes / adaptadores de canal). De esta forma:

- La lógica de notificación queda desacoplada del controlador y de la persistencia.
- Se puede añadir o quitar canales de notificación (EmailObserver, PushObserver, InAppObserver) sin cambiar el Subject.
- La notificación se puede ejecutar sincrónica (para pruebas) o asíncronamente via cola de mensajes (Recomendado en producción).
- Facilita la gestión de suscripciones: los observers se registran (attach) y se eliminan (detach) dinámicamente.

Componentes propuestos:

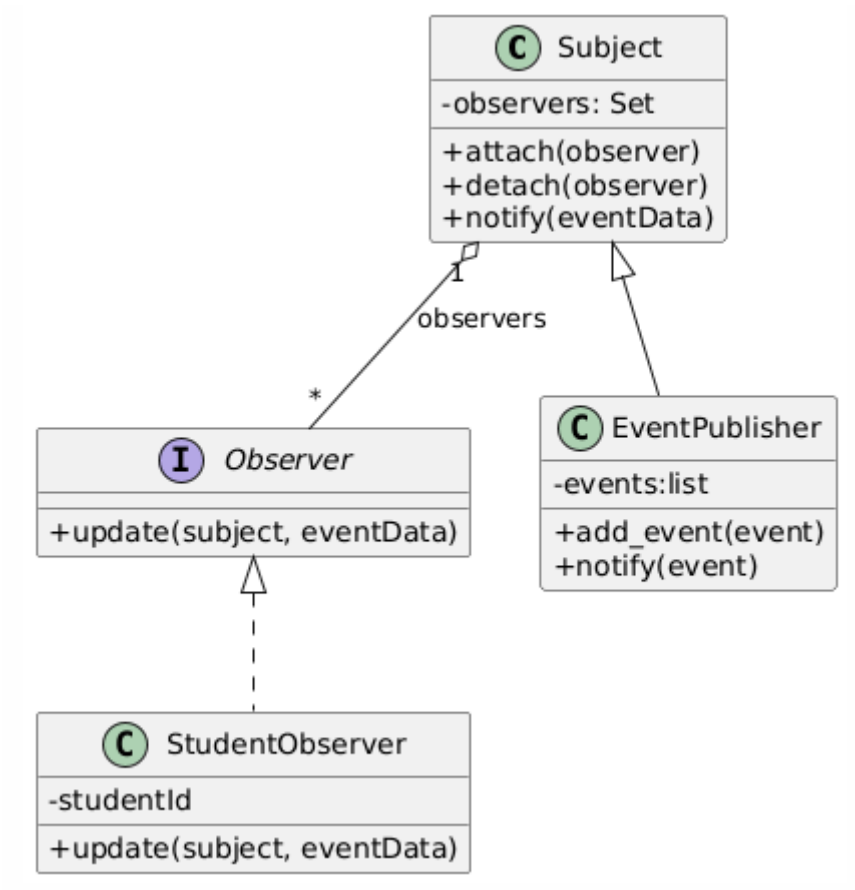
- `EventPublisher` — ConcreteSubject: módulo responsable de persistir eventos y disparar notificaciones.
- `StudentObserver` — ConcreteObserver: representa a un estudiante suscrito (puede delegar a `EmailObserver`, `PushObserver`).
- `NotificationService` — servicio que encapsula la lógica de envío (SMTP, FCM, WebPush).
- `SubscriptionRepository` — repositorio que devuelve la lista de suscriptores de un club.
- `Worker / MQ` — (opcional) cola y workers para procesar notificaciones en segundo plano.

Participantes / roles y responsabilidades

- Subject (Observable)
    - Métodos: `attach(observer)`, `detach(observer)`, `notify(eventData)`.
    - Responsabilidad: mantener la lista de observadores y llamar `update()` cuando cambia el estado (nuevo evento).
- Observer (interfaz)

    - Método: `update(subject, eventData)`.
    - Responsabilidad: recibir la notificación y actuar (enviar email, push, in-app).
- ConcreteSubject (EventPublisher)
    - Función: persistir evento, obtener suscriptores, registrar/adjuntar observers y notificar.
- ConcreteObserver (StudentObserver, EmailObserver, PushObserver, InAppObserver)
    - Función: implementar `update()` y usar `NotificationService` para enviar el aviso.

Diagrama de clases:



Pseudocódigo en python

```python
# Interfaz Observer
class Observer:
    def update(self, subject, event_data):
        pass

# Subject (Observable)
class Subject:
    def __init__(self):
        self._observers = set()
    def attach(self, observer: Observer):
        self._observers.add(observer)
    def detach(self, observer: Observer):
        self._observers.discard(observer)
    def notify(self, event_data=None):
        for obs in list(self._observers):
            try:
                obs.update(self, event_data)
            except Exception:
                # log y continuar
                pass

# ConcreteSubject (EventPublisher)
class EventPublisher(Subject):
    def __init__(self, subscription_repo, notification_service):
        super().__init__()
        self.subscription_repo = subscription_repo
        self.notification_service = notification_service

    def add_event(self, event):
        # persistir (dentro de transacción)
        EventsRepository.save(event)
        # post-commit: obtener suscriptores
        subscribers =
self.subscription_repo.get_subscribers(event.club_id)
        for s in subscribers:
            obs = StudentObserver(s.id, self.notification_service)
            self.attach(obs)
        # notificar (puede delegar a MQ para workers)
        self.notify(event)

# ConcreteObserver (StudentObserver)
class StudentObserver(Observer):
    def __init__(self, student_id, notification_service):
```

```
        self.student_id = student_id
        self.notification_service = notification_service
    def update(self, subject, event_data):
        self.notification_service.send_email(self.student_id,
event_data)
```

El patrón Observer es una solución apropiada y natural para el requisito de notificar a suscriptores en **ÁgoraUN**. Permite un diseño modular, extensible y alineado con buenas prácticas arquitectónicas (separación de responsabilidades). Lo ideal sería implementarlo con notificaciones asíncronas (MQ + workers) y con tests automatizados que verifiquen tanto la lógica de suscripción como la entrega de mensajes.

Referencias

- https://refactoring.guru/es/design-patterns/observer