PROYECTO FINAL
PATRONES
Entrega 04

Elaborado por:
Error 404

Integrantes:
Bermudez Guaqueta Tomas Alejandro
Cely Infante Daniel Alfonso
Gracia Pinto Daniel Alonso
Herrera Novoa David Alejandro

Docente:
Oscar Eduardo Alvarez Rodriguez
Ingeniería de Software I

UNIVERSIDAD NACIONAL DE COLOMBIA
FACULTAD DE INGENIERIA
Bogotá, 2025

I. PATRÓN SINGLETON

1. Resumen y Propósito

En el contexto de Django, esto se utilizará **"Singleton Model"**. El objetivo es garantizar que un modelo específico, que usaremos para la configuración global del sitio, tenga una sola instancia (una única fila) en la base de datos.

Esta implementación nos permite centralizar todas las configuraciones globales de la plataforma SAFE (ej. intentos de examen por defecto) en un modelo editable desde el panel de administración, en lugar de tener valores "hardcodeados" en config/settings.py.

La implementación se divide en dos partes:

21. Una clase base abstracta (SingletonModel) que contiene la lógica del patrón.
22. Un modelo concreto (SiteConfiguration) que hereda esta lógica y define nuestros campos de configuración.

2. El Modelo Base Abstracto: SingletonModel

Esta es una clase de utilidad abstracta y reutilizable. El lugar ideal para este código es una nueva app core dedicada a la lógica central y utilidades compartidas.

Acción: Crear una app core (si no existe) y añadirla a INSTALLED_APPS.

Ubicación: SAFE/core/models.py

```python
# SAFE/core/models.py
from django.db import models

class SingletonModel(models.Model):
    """
    Abstract base class implementing the Singleton design pattern.

    Ensures that any model inheriting from this class will have only
    one instance in the database, conventionally with pk=1.
    """

    class Meta:
        # Define this class as abstract, so Django won't
        # create a 'core_singletonmodel' table.
        abstract = True

    def save(self, *args, **kwargs):
        """
        Override the default save method.
```

```
        Forces the object to always be saved with pk=1.
        """
        self.pk = 1
        super(SingletonModel, self).save(*args, **kwargs)

    def delete(self, *args, **kwargs):
        """
        Override the delete method to prevent deletion
        of the unique instance.
        """
        # Optionally: raise PermissionDenied("Cannot delete configuration.")
        pass # Simply ignore the delete request.

    @classmethod
    def load(cls):
        """
        Global access point (equivalent to 'getInstance()').

        Uses 'get_or_create' to fetch the instance with pk=1.
        If the instance doesn't exist (e.g., first run after
        migrations), it creates it automatically with default values.

        Returns:
            (cls): The unique instance of the settings model.
        """
        # 'obj' is the model instance, 'created' is a boolean.
        obj, created = cls.objects.get_or_create(pk=1)
        return obj
```

3. Implementación: SiteConfiguration

Este es el modelo concreto que almacenará nuestras configuraciones. Dado que tienes una app config que ya maneja settings.py, este es el lugar semántico perfecto para un modelo de configuración de base de datos.

Ubicación: SAFE/config/models.py

Python

```
# SAFE/config/models.py
from django.db import models
from core.models import SingletonModel # <-- Import from our 'core' app
```

```python
class SiteConfiguration(SingletonModel):
    """
    Singleton model to store global settings for the
    SAFE e-learning platform.

    This table will contain a single row (pk=1) accessible
    from anywhere in the project to query global business rules.
    """

    site_name = models.CharField(
        max_length=150,
        default='SAFE:Sistema Académico de Formación Empresarial',
        help_text="Tu mejor aliado en la formación empresarial."
    )

    maintenance_mode = models.BooleanField(
        default=False,
        help_text="Check this box to put the site in maintenance mode. "
                  "Regular users ('colaborador') will be locked out."
    )

    allow_new_enrollments = models.BooleanField(
        default=True,
          help_text="Globally allow or block new enrollments in courses or
paths."
    )

    default_max_exam_tries = models.PositiveSmallIntegerField(
        default=3,
        help_text="Default number of attempts for an 'exam' if it does not "
                  "specify its own limit ('max_tries' in 'exam' table)."
    )

    min_passing_score = models.PositiveSmallIntegerField(
        default=60,
          help_text="Global minimum score (out of 100) to pass an 'exam' or
'assignment'."
    )

    def __str__(self):
        # Text displayed in the Django admin
        return "Configuración Global de SAFE"

    class Meta:
        # Human-readable name for the admin panel
        verbose_name = "Configuración Global del Sitio"
        verbose_name_plural = "Configuración Global del Sitio"
```

4. Habilitación en el Panel de Administración

Para hacer este modelo editable por los administradores (supervisor, analistaTH), lo registramos en el admin.py de la app config.

Ubicación: SAFE/config/admin.py

```python
# SAFE/config/admin.py
from django.contrib import admin
from .models import SiteConfiguration # <-- Import from local models.py

@admin.register(SiteConfiguration)
class SiteConfigurationAdmin(admin.ModelAdmin):
    """
    Admin configuration for the Singleton model.

    We disable 'add' and 'delete' permissions in the UI
    to reinforce the Singleton pattern.
    """
    list_display = (
        'site_name',
        'maintenance_mode',
        'allow_new_enrollments',
        'default_max_exam_tries'
    )

    # Prevent adding new instances from the admin UI
    def has_add_permission(self, request):
        return False

    # Prevent deleting the instance from the admin UI
    def has_delete_permission(self, request, obj=None):
        return False
```

5. Guía de Uso para Desarrolladores

Para consumir la configuración en cualquier parte del proyecto (vistas, serializadores, etc.), siempre use el método .load() para garantizar un acceso seguro.

INCORRECTO (No hacer):

```python
# This code is fragile and will crash if the pk=1 instance
# has not been created yet.
from config.models import SiteConfiguration
config = SiteConfiguration.objects.get(pk=1)
```

CORRECTO (Uso oficial):

```python
# This is safe. It gets the instance or creates it
# with defaults if it doesn't exist.
from config.models import SiteConfiguration
config = SiteConfiguration.load()
```

Ejemplo de Uso: Vista de un Examen

A continuación, un ejemplo de cómo usar la configuración en la app courses para verificar reglas de negocio.

Ubicación: SAFE/courses/views.py (o SAFE/enrollments/views.py donde corresponda)

```python
# SAFE/courses/views.py
from django.shortcuts import render
from django.http import Http404
from .models import Exam # Assuming Exam model is in 'courses'
from enrollments.models import ContentProgress # Assuming this structure
from config.models import SiteConfiguration # 1. Import the Singleton

def attempt_exam_view(request, exam_id):

    # 2. Load the global config using the Singleton method
    config = SiteConfiguration.load()

    # 3. Apply global business rules

    # Rule 1: Check for maintenance mode
```

```
if config.maintenance_mode and not request.user.is_staff:
    # (Assuming 'supervisor'/'analistaTH' are staff users)
    return render(request, 'maintenance.html')

try:
    exam = Exam.objects.get(pk=exam_id)

    # ... logic to get user's enrollment ...
    user_enrollment = ...

    # Count previous attempts
    previous_attempts = ContentProgress.objects.filter(
        course_inscription=user_enrollment,
        content__exam_id=exam.id
    ).count()

    # Rule 2: Use global config as a fallback
    # If the 'exam' has its own 'max_tries', use it.
    # If it's NULL, use the global value from the Singleton.
    max_tries = exam.max_tries or config.default_max_exam_tries

    if previous_attempts >= max_tries:
        return render(request, 'error.html', {
            'message': f"Límite de {max_tries} intentos alcanzado."
        })

    # ... rest of the view logic to present the exam ...

except Exam.DoesNotExist:
    raise Http404("Examen no encontrado")
```

II. PATRÓN OBSERVER

1. Resumen y Propósito

El Patrón Observer (Observador) es un patrón de diseño de comportamiento que define una dependencia de uno-a-muchos entre objetos. Cuando un objeto (el "Sujeto") cambia su estado, todos sus dependientes (los "Observadores") son notificados y actualizados automáticamente.

El propósito principal de este patrón es promover un bajo acoplamiento (loose coupling) entre el sujeto y sus observadores. El Sujeto no necesita saber nada sobre quiénes son sus Observadores o qué hacen; simplemente mantiene una lista de ellos y les notifica cuando ocurre un evento.

2. El Patrón Observer en Django: "Signals"

Django proporciona una implementación elegante y robusta del patrón Observer lista para usar: el sistema de "Signals" (Señales).

En este paradigma:

- **El Sujeto (Subject):** Es el "emisor" de la señal (el *Sender*). Generalmente, es un modelo de Django (ej. Course, AppUser).
- **El Evento (Event):** Es la propia "Señal" (Signal) que se emite. Django provee señales predefinidas para los eventos más comunes del ciclo de vida de un modelo, como pre_save, post_save, pre_delete, y post_delete.
- **El Observador (Observer):** Es el "receptor" de la señal (el *Receiver*). Este es simplemente una función de Python que se "suscribe" a una señal específica de un emisor específico.

Esta arquitectura es la forma idiomática ("Django way") de implementar el patrón Observer y es ideal para mantener nuestro código limpio y desacoplado, lo cual es fundamental para el proyecto SAFE.

3. Escenario de Uso

Un escenario posible podría ser el siguiente:

> **"**Cuando un Analista TH publica un nuevo curso (es decir, lo crea y lo marca como 'active'), el sistema debe notificar automáticamente a los colaboradores de su equipo para que estén al tanto del nuevo contenido**."**

Análisis del escenario:

- **Sujeto (Subject):** El modelo Course.
- **Evento (Event):**El momento exacto después de que una nueva instancia de Course se guarda en la base de datos (post_save).
- **Observador (Observer):** Una función receptora (un *receiver*) que definiremos dentro de tu app notifications.
- **Acción del Observador:**
    1. Verificar si el curso es *nuevo* (created=True) y su *estado* es 'active'.
    2. Identificar al creador (created_by), que es el 'Analista TH'.
    3. Buscar los equipos (Team) que este analista supervisa.
    4. Obtener todos los usuarios (TeamUser) que pertenecen a esos equipos y que tienen el rol de 'colaborador'.
    5. Crear un objeto Notification para cada uno de esos colaboradores.

Justificación (¿Por qué Signals?): Usar Signals (la implementación de Observer en Django) es crucial aquí. La alternativa sería poner toda esta lógica de notificación dentro del método .save() del modelo Course o en la vista que crea el curso.

Eso crearía un acoplamiento fuerte. La app courses se vería forzada a importar modelos de teams, accounts y notifications. Esto viola el Principio de Responsabilidad Única (SRP).

Con el patrón Observer, la app courses simplemente "anuncia" que un curso fue creado. La app notifications es la que decide "escuchar" ese anuncio y reaccionar. Las apps permanecen limpias y desacopladas.

4. Implementación Detallada

A continuación mostraremos cómo se realizaría la implementación de este flujo usando un receptor de señal en la app notifications.

Paso 1: Crear el Observador (El Receptor)

Creamos un nuevo archivo signals.py dentro de la app notifications. Aquí es donde reside nuestra lógica de "Observador".

SAFE/notifications/signals.py (Archivo nuevo)

Python

```python
# Django imports for Signals
from django.db.models.signals import post_save
from django.dispatch import receiver

# Import models from other apps
# These are the models we need to read from
from courses.models import Course
from teams.models import Team, TeamUser
from accounts.models import AppUser # Assuming AppUser is in accounts

# This is the model we will write to
from .models import Notification

@receiver(post_save, sender=Course)
def notify_team_on_new_course(sender, instance, created, **kwargs):
    """
    Observer (Receiver) that listens for 'post_save' signals
    from the 'Course' model.

    'instance' is the Course object that was saved.
    'created' is a boolean (True if this was a new record).
    """

    # 1. Check if the event matches our requirement
    if created and instance.status == 'active':
```

```
    # 2. Identify the creator (the 'Analista TH' or 'Supervisor')
    creator = instance.created_by
    if not creator:
        return # Safety check

    # 3. Find teams managed by this creator
    # We query the Team model using the creator
    teams_managed = Team.objects.filter(supervisor=creator)
    if not teams_managed.exists():
        return # This person manages no teams, so nothing to do

    # 4. Find all users in those teams
    # We get all TeamUser entries linked to those teams
    team_members_links = TeamUser.objects.filter(
        team__in=teams_managed
    )

    # 5. Get the actual collaborator user objects
    # We fetch the AppUser objects for those links,
    # ensuring they are 'colaborador' and not the creator.
    collaborator_ids = team_members_links.values_list(
        'app_user_id',
        flat=True
    )

    collaborators_to_notify = AppUser.objects.filter(
        id__in=collaborator_ids,
        role='colaborador'
    ).exclude(
        id=creator.id # Don't notify the creator
    ).distinct()


    # 6. Create notifications in bulk (very efficient)
    message = (
        f"¡Nuevo curso disponible! "
        f"'{instance.name}' ha sido publicado."
    )

    notifications_to_create = [
        Notification(user=user, message=message)
        for user in collaborators_to_notify
    ]

    if notifications_to_create:
        Notification.objects.bulk_create(notifications_to_create)
```

Paso 2: Conectar el Observador

Ahora, debemos decirle a Django que cargue este archivo signals.py cuando se inicie la aplicación. La forma moderna de hacerlo es en el archivo apps.py de la app notifications.

SAFE/notifications/apps.py (Modificar este archivo)

```python
from django.apps import AppConfig

class NotificationsConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'notifications'

    def ready(self):
        """
        This method is called when the app is ready.
        We import our signals file here to connect the receivers.
        """
        import notifications.signals # This line is new
```

Paso 3: Informar a Django que use esta Configuración

Finalmente, asegúrate de que Django use tu NotificationsConfig personalizada.

SAFE/notifications/__init__.py (Archivo existente, se debe modificar)

```python
# This line tells Django to use your AppConfig
default_app_config = 'notifications.apps.NotificationsConfig'
```

Nota: Para implementar este escenario, necesitamos un lugar donde almacenar esas notificaciones. El esquema de base de datos no incluye una tabla Notification. Por lo cual se debería construir esta tabla.

5. Resultado y Ventajas

Con esta implementación:

1. **Desacoplamiento Total:** La app courses (y su models.py) está 100% limpia. No sabe nada sobre teams o notifications. Simplemente guarda un curso.
2. **Responsabilidad Única:** La lógica de *quién* debe ser notificado y *cómo* se crea la notificación vive enteramente dentro de la app notifications, que es su lugar correcto.
3. **Extensibilidad:** Si en el futuro también se quiere enviar un email cuando se publica un curso, no tienes que tocar courses *ni* notifications. Simplemente se puede crear *otro receptor* (quizás en accounts/signals.py) que también escuche la señal post_save de Course y se encargue de la lógica de envío de correos.