DOCUMENTACIÓN DE PATRONES DE DISEÑO

Proyecto: Hola Mundo Full-Stack con Django y PostgreSQL

Introducción:

Este documento presenta los patrones de diseño utilizados en el proyecto *Hola Mundo Full-Stack con Django y PostgreSQL*. En particular, se describen el patrón Singleton (implementado) y el patrón Factory Method (propuesto), ambos pertenecientes a la categoría de patrones creacionales.

- **Patrón Singleton (Implementado)**

El patrón de diseño Singleton tiene como propósito restringir la instanciación de una clase a un único objeto y proporcionar un punto de acceso global a esta instancia. Pertenece a la categoría de patrones creacionales y resulta útil cuando se necesita exactamente un objeto para coordinar acciones en todo el sistema.

En el contexto del proyecto, se implementó el patrón Singleton para el sembrado inicial de datos en la base de datos. En lugar de gestionar manualmente la conexión (lo cual sería un anti-patrón en Django), se creó una clase **InitialDataSeeder** que garantiza que la creación del mensaje 'Hola Mundo' en la base de datos ocurra una sola vez al iniciar el servidor.
Este enfoque asegura que el registro con ID=1 exista siempre, y evita duplicaciones incluso en entornos con múltiples hilos.

Código :

```python
class InitialDataSeeder:
    _instance = None
    _lock = threading.Lock()
    _seeded = False

    def __new__(cls) -> Self:
        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    print("Creando instancia Singleton del Seeder...")
                    cls._instance: Self = super().__new__(cls)
        return cls._instance

    def seed_initial_data(self) -> None:
        if not self._seeded:
            with self._lock:
                if not self._seeded:
                    from .models import Message
                    try:
                        Message.objects.get(pk=1)
                        print("Mensaje inicial ya existe.")
                    except Message.DoesNotExist:
                        Message.objects.create(pk=1, content="Hola Mundo desde Django + BD! 🐘")
                        print("Mensaje inicial creado.")
                    self._seeded = True

initial_seeder = InitialDataSeeder()
```

```python
class HelloappConfig(AppConfig):
    default_auto_field: str = 'django.db.models.BigAutoField'
    name = 'helloapp'

    def ready(self) -> None:
        from .initial_seeder import initial_seeder
        initial_seeder.seed_initial_data()
```

se ejecuta automáticamente al iniciar la aplicación mediante el método **ready()** definido en **apps.py**, cumpliendo así con el requisito de inicialización única de datos.

Patrón Factory Method (Propuesto)

El patrón Factory Method permite definir una interfaz para crear objetos en una superclase, dejando que las subclases decidan qué clase instanciar. Esto facilita la extensión del código sin modificar las clases existentes, promoviendo el principio de abierto/cerrado (OCP).
En el proyecto se propone su uso para la generación de reportes en diferentes formatos (PDF, CSV, XLSX). La clase **ReportFactory** decide qué tipo de generador instanciar según el formato solicitado, evitando estructuras condicionales rígidas dentro del servicio.

```python
class IReportGenerator:
    def generate(self, data) -> Any:
        raise NotImplementedError

class PDFGenerator(IReportGenerator):
    def generate(self, data) -> Literal['reporte.pdf']:
        print("Generando PDF...")
        return "reporte.pdf"

class CSVGenerator(IReportGenerator):
    def generate(self, data) -> Literal['reporte.csv']:
        print("Generando CSV...")
        return "reporte.csv"

class ReportFactory:
    @staticmethod
    def get_generator(format: str) -> IReportGenerator:
        if format == 'pdf':
            return PDFGenerator()
        elif format == 'csv':
            return CSVGenerator()
        else:
            raise ValueError(f"Formato '{format}' no soportado")

```

De esta forma, el servicio de generación de reportes no necesita modificarse al incorporar nuevos formatos. Basta con crear una nueva clase concreta e incorporarla en la fábrica.
Conclusión

Conclusión

Ambos patrones de diseño (Singleton y Factory Method) fortalecen la arquitectura del proyecto al garantizar consistencia en la inicialización de datos y extensibilidad en la creación de objetos. El patrón Singleton se implementó de manera segura y controlada en el contexto de Django, mientras que el Factory Method se propuso para un futuro módulo de generación de reportes, promoviendo un código limpio, modular y fácil de mantener.