Patrones de Diseño SincroHealth

grupo 2 - #abrancupos

Integrantes:

- *Salazar Carvajal Omar Alejandro*
- *Pajaro Sanchez Nicolas*
- *Diaz Acosta Nathalia Milena*
- *Camargo Barrera Samuel Francisco*

SINGLETON

Nuestro proyecto usa como framework a ***Django***. Django no necesita Singleton para las conexiones de la base de datos, ya que estas se gestionan de forma automática. Mediante el uso de ***ORM*** y ***señales*** se garantiza la unicidad y acceso global. Podemos observar el uso de estos en el siguiente fragmento de código de nuestro proyecto:

```
class HistorialMedico(models.Model):
    idhistorial = models.AutoField(primary_key=True)
    idpaciente = models.OneToOneField('Paciente', on_delete=models.CASCADE)
    recetas_json = models.JSONField(blank=True, default=list)
    fecha = models.DateField(auto_now=True)

    def actualizar_recetas(self):
        recetas = self.idpaciente.recetamedica_set.all().values(
            'idrecetas_medicas',
            'diagnostico',
            'medicamentos',
            'indicaciones',
            'idcitas_medicas'
        )
        self.recetas_json = list(recetas)
        self.save()
```

El **ORM** toma el modelo ***HistorialMedico*** y lo traduce a ***SQL*** creando la tabla en la base de datos. También podemos apreciar el uso del módulo en consultas y relaciones automáticas en ***`recetas = self.idpaciente.recetamedica_set.all().values(...)`***

Las **señales** nos permiten ejecutar código automáticamente cuando ocurre un evento, un uso de esto en nuestro proyecto lo vemos en el siguiente fragmento. Específicamente en el decorador ***`@receiver(post_save, sender='polls.RecetaMedica')`***:

```
@receiver(post_save, sender='polls.RecetaMedica')
def actualizar_historial_al_guardar(sender, instance, **kwargs):
    paciente = instance.idpaciente
    historial, creado =
HistorialMedico.objects.get_or_create(idpaciente=paciente)
```

```
    historial.actualizar_recetas()
```

Aunque gracias a estas herramientas de Django no es necesario el uso de Singleton para nuestro proyecto, podemos implementarlo. En el siguiente fragmento de código veremos como se podría aplicar a un proyecto Django:

```
# app/configuracion.py
class SingletonConfig:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(SingletonConfig, cls).__new__(cls)
            # Inicializa tus datos aquí, por ejemplo, cargando un archivo
             cls._instance.config = {"db_url": "..."}
        return cls._instance
```

Se crea la clase en un módulo separado que luego se podrá importar en la parte del proyecto que se necesite:

```
# app/views.py
from .config_manager import SingletonConfig

def my_view(request):
    config = SingletonConfig()
    # Ahora puedes usar la instancia de config
    return JsonResponse(config.config)
```

OBSERVER

La principal función de este patrón de diseño es notificar a las clases de algún evento. En ***Django*** el principal mecanismo que se utiliza para cumplir con esta función son las ***señales***, las cuales ya mencionamos en la sección del patrón Singleton. Cuando ocurre un evento las señales notifican a receptores para que ejecuten una acción. En el siguiente fragmento de código es un ejemplo de cómo aplicamos las señales a nuestro proyecto:

```
@receiver(post_save, sender='polls.RecetaMedica')
def actualizar_historial_al_guardar(sender, instance, **kwargs):
    paciente = instance.idpaciente
    historial, creado =
HistorialMedico.objects.get_or_create(idpaciente=paciente)
    historial.actualizar_recetas()
```

```
@receiver(post_delete, sender='polls.RecetaMedica')
def actualizar_historial_al_eliminar(sender, instance, **kwargs):
    paciente = instance.idpaciente
    historial =
HistorialMedico.objects.filter(idpaciente=paciente).first()
    if historial:
        historial.actualizar_recetas()
```

Podemos observar dos señales: ***post_save*** y ***post_delate***. Después de agregar una receta médica, se ejecutará ***actualizar_historial_al_guardar*** como receptor de ***post_save***. Guardando la receta médica creada en la columna perteneciente a todas las recetas médicas que tiene un paciente. Así mismo se ejecutará ***actualizar_historial_al_eliminar*** como receptor de ***post_delete***, eliminando de la lista una receta médica si ésta fue eliminada de la ***RecetaMedica.***

STATE

Este patrón de diseño permite a una clase alterar su comportamiento cuando su estado interno cambia. Nuestra idea para aplicarla en nuestro proyecto sería implementarla para los estados de las citas médicas (pendiente, cancelada, realizada). Para esto tenemos que crear un nuevo archivo en el proyecto para poner los diferentes estados en donde la estructura básica del código sería:

```
class Pendiente: #estado predeterminado
    def agendar(self, cita):
        print("La cita ya está agendada y en estado pendiente.")
        return False
    def cancelar(self, cita):
        cita.estado_cita = "Cancelada"
        cita.save()
        print(f"Cita {cita.idcitas_medicas} cancelada.")
        return True
    def realizar(self, cita):
        cita.estado_cita = "Realizada"
        cita.save()
        print(f"Cita {cita.idcitas_medicas} realizada.")
        return True

class Cancelada:
    def agendar(self, cita):
        print("No se puede agendar una cita cancelada.")
        return False
    def cancelar(self, cita):
        print("La cita ya está cancelada.")
```

```
        return False
    def realizar(self, cita):
        print("No se puede realizar una cita cancelada.")
        return False

class Realizada:
    def agendar(self, cita):
        print("No se puede agendar una cita ya realizada.")
        return False
    def cancelar(self, cita):
        print("No se puede cancelar una cita ya realizada.")
        return False
    def realizar(self, cita):
        print("La cita ya está realizada.")
        return False
#mapeo para asociar clave con valor
MAPA_ESTADOS = {
    "Pendiente": Pendiente(),
    "Cancelada": Cancelada(),
    "Realizada": Realizada(),
}
```

STRATEGY

Este patrón de diseño permite crear una familia de algoritmos para hacer nuestros objetos intercambiables. Nuestra idea para implementarlo a nuestro proyecto sería para las notificaciones y recordatorios. El primer paso sería crear un nuevo archivo al proyecto para las notificaciones, la estructura básica del código para poder implementar este patrón de diseño sería:

```
class NotificacionStrategy:
    def enviar(self, cita):
        raise NotImplementedError

class EmailNotificacion(NotificacionStrategy):
    def enviar(self, cita):
        ...
        print(f"Enviando correo de recordatorio para la cita
{cita.idcitas_medicas}.")
```

Luego en el archivo ***views.py*** se pondrá la función para poder crear la estrategia que se requiera para cada caso específico.