Esta es la pieza final para tener un sistema de administración responsable. Un "Logger de Auditoría" no solo es para seguridad, sino para **trazabilidad**: saber exactamente *quién* cambió *qué* y *cuándo*.

Aquí tienes la implementación en **Python (FastAPI)**.

1. El Modelo de Base de Datos (`models.py`)

Primero, definimos la tabla en SQLAlchemy para que Python pueda escribir en ella. Asegúrate de tener `JSON` (o `JSONB` si usas PostgreSQL) importado.

Python
```
from sqlalchemy import Column, String, DateTime, ForeignKey
from sqlalchemy.dialects.postgresql import JSONB # Usar JSON si es SQLite/MySQL
from sqlalchemy.sql import func
from uuid import uuid4

# Asumiendo que 'Base' viene de tu configuración principal
class AuditLog(Base):
    __tablename__ = "audit_logs"

    id = Column(String, primary_key=True, default=lambda: str(uuid4()))
    actor_id = Column(String, ForeignKey("users.id"))  # El Admin que ejecuta
    target_id = Column(String, ForeignKey("users.id"), nullable=True) # El usuario afectado
    action_type = Column(String, index=True) # Ej: "ROLE_CHANGE", "BAN_USER"
    details = Column(JSONB) # Diccionario flexible para guardar metadatos
    ip_address = Column(String, nullable=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

2. La Función del Logger (`audit_service.py`)

No queremos escribir código de inserción en cada endpoint. Creamos una función reutilizable.
Truco Pro: Usaremos `BackgroundTasks` de FastAPI para que el log se guarde *después* de responder al usuario, así no ralentizamos la API.

Python
```
from sqlalchemy.orm import Session
from fastapi import Request

def create_audit_log(
    db: Session,
    actor_id: str,
```

```
    action_type: str,
    target_id: str = None,
    details: dict = None,
    ip_address: str = "0.0.0.0"
):
    """
    Registra una acción crítica en la base de datos de auditoría.
    """
    new_log = AuditLog(
        actor_id=actor_id,
        target_id=target_id,
        action_type=action_type,
        details=details or {},
        ip_address=ip_address
    )
    db.add(new_log)
    db.commit()
    # No hace falta db.refresh() porque no necesitamos leerlo de vuelta inmediatamente

# Wrapper para capturar la IP automáticamente desde el Request
def log_action_background(
    background_tasks,
    db_session_factory, # Pasamos el factory porque la sesión original puede cerrarse
    request: Request,
    actor_id: str,
    action_type: str,
    target_id: str = None,
    details: dict = None
):
    client_ip = request.client.host

    # Definimos la tarea que correrá en segundo plano
    def _task():
        db = db_session_factory() # Abrir nueva sesión efímera
        try:
            create_audit_log(db, actor_id, action_type, target_id, details, client_ip)
        finally:
            db.close()

    background_tasks.add_task(_task)
```

3. Implementación en un Endpoint de Admin (Ejemplo Real)

Vamos a aplicar esto al endpoint crítico de **"Cambiar Rol de Usuario"** o **"Revocar Acceso"**.

Python

```
from fastapi import APIRouter, Depends, HTTPException, BackgroundTasks, Request
from main import get_db, SessionLocal, User, require_role # Tus importaciones previas

router = APIRouter(prefix="/admin", tags=["Administración"])

class RoleChangeRequest(BaseModel):
    target_email: str
    new_role: str # Ej: "Operador"

@router.post("/change-role")
def admin_change_user_role(
    payload: RoleChangeRequest,
    request: Request,
    background_tasks: BackgroundTasks,
    current_user: User = Depends(get_current_user), # El Admin
    db: Session = Depends(get_db)
):
    # 1. Seguridad: Solo Admins
    require_role(current_user, "Admin")

    # 2. Buscar al usuario objetivo
    target_user = db.query(User).filter(User.email == payload.target_email).first()
    if not target_user:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")

    # 3. Evitar auto-bloqueo (Regla de negocio)
    if target_user.id == current_user.id:
        raise HTTPException(status_code=400, detail="No puedes cambiar tu propio rol")

    # 4. LA ACCIÓN: Cambiar el rol (Lógica simplificada)
    # Aquí deberías buscar el ID del rol en la tabla Roles, etc.
    old_roles = [r.name for r in target_user.roles]
    # ... lógica para actualizar target_user.roles ...

    db.commit()

    # 5. EL LOGGER DE AUDITORÍA (Aquí ocurre la magia)
    # Registramos QUÉ pasó sin bloquear la respuesta rápida al Admin
    log_details = {
        "previous_roles": old_roles,
        "new_role_requested": payload.new_role,
```

```
    "reason": "Solicitud administrativa manual"
}

log_action_background(
    background_tasks=background_tasks,
    db_session_factory=SessionLocal, # Importante: pasar el creador de sesiones
    request=request,
    actor_id=current_user.id,     # ¿Quién lo hizo? El Admin
    target_id=target_user.id,     # ¿A quién? Al usuario objetivo
    action_type="ROLE_CHANGE",    # ¿Qué hizo?
    details=log_details           # Detalles técnicos
)

return {"msg": f"Rol de usuario {payload.target_email} actualizado y acción auditada."}
```

¿Qué acabas de lograr?

Con este código, cada vez que un Admin haga algo delicado, tu base de datos guardará silenciosamente:

- **Responsabilidad:** `actor_id` (Fue el Admin Dr. Nexo).
- **Contexto:** `ip_address` (Lo hizo desde la IP de la oficina 192.168.x.x).
- **Detalle Forense:** `details` (Antes el usuario era "Viewer", ahora es "Admin").

Esto cumple perfectamente con el requisito de **"Segregación y Monitoreo"**. Si algo sale mal, tienes la "Caja Negra" del sistema para investigar.