src/main.py

```python
from fastapi import FastAPI, Depends, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from sqlalchemy.orm import Session
from fastapi.responses import JSONResponse
from fastapi.requests import Request
from src import models
from src.database import SessionLocal
from src.schemas_movimientos import MovimientoCreate
from src.schemas_movimientos import MovimientoUpdate
from src.schemas_deudas import DeudaCreate
from src.schemas_deudas import DeudaUpdate
from src.schemas_descripciones import DescripcionCreate
from src.schemas_descripciones import DescripcionUpdate
from datetime import date
import traceback
from fastapi import Query
from fastapi import APIRouter
from datetime import datetime, date
from sqlalchemy import extract, func
from src.models import Deuda
# from decimal import Decimal
from decimal import Decimal, InvalidOperation
from typing import List,Optional
#agrego este import para agregar la columna deuda_saldo_antes  en la  tabla movimientos
from sqlalchemy import DECIMAL
from sqlalchemy.orm import Session

#agregar conexion directa para consultas personalizadas
from src.database import engine
from sqlalchemy import text

from fastapi.middleware.cors import CORSMiddleware


app = FastAPI()

@app.exception_handler(Exception)
async def all_exception_handler(request: Request, exc: Exception):
    traceback.print_exc()
    return JSONResponse(status_code=500, content={"detail": str(exc)})

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=False,  # ← ESTO ES CRÍTICO
    allow_methods=["*"],
    allow_headers=["*"],
)
```

```python
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

#Endpoints para páginas de movimientos de ingresos
@app.post("/ingresos")
def crear_ingreso(movimiento: MovimientoCreate, db: Session = Depends(get_db)):
    return crear_movimiento(movimiento, db)


@app.post("/movimientos")
def crear_movimiento(movimiento: MovimientoCreate, db: Session = Depends(get_db)):
    # Convertir monto a Decimal
    monto_pago = Decimal(str(movimiento.monto))

    # Asegurarse de que deuda_id sea int o None
    deuda_id = int(movimiento.deuda_id) if movimiento.deuda_id else None

    # Crear el movimiento
    nuevo_mov = models.Movimiento(
        descripcion_id=movimiento.descripcion_id,
        monto=monto_pago,
        fecha=movimiento.fecha or date.today(),
        deuda_id=deuda_id
    )
    db.add(nuevo_mov)

    # Actualizar deuda si corresponde
    if deuda_id:
        deuda = db.query(models.Deuda).filter(models.Deuda.id == deuda_id).first()
        if deuda:
            monto_deuda = Decimal(str(deuda.monto))
            monto_pagado_actual = Decimal(str(deuda.monto_pagado or '0.00'))

            # Sumar pago
            deuda.monto_pagado = (monto_pagado_actual +
monto_pago).quantize(Decimal('0.01'))

            # Calcular saldo pendiente
            deuda.saldo_pendiente = (monto_deuda -
deuda.monto_pagado).quantize(Decimal('0.01'))

            # Marcar pagado si saldo pendiente <= 0
            if deuda.saldo_pendiente <= Decimal('0.00'):
                deuda.pagado = True
                deuda.saldo_pendiente = Decimal('0.00')
```

```
    db.commit()
    db.refresh(nuevo_mov)

    # Preparar respuesta con info de descripción
    descripcion_nombre = nuevo_mov.descripcion.nombre if nuevo_mov.descripcion else
""

    return {
        "mensaje": "Movimiento registrado correctamente",
        "movimiento": {
            "id": nuevo_mov.id,
            "descripcion_id": nuevo_mov.descripcion_id,
            "categoria": descripcion_nombre,
            "monto": str(nuevo_mov.monto),
            "fecha": nuevo_mov.fecha.isoformat(),
            "deuda_id": nuevo_mov.deuda_id
        }
    }


# OPCIONAL: Mantener endpoints específicos que redirijan al unificado
@app.put("/ingresos/{ingreso_id}")
def actualizar_ingreso(ingreso_id: int, movimiento: MovimientoCreate, db: Session =
Depends(get_db)):
    return actualizar_movimiento(ingreso_id, movimiento, db)


# 3. ACTUALIZAR: Endpoint unificado de movimientos con mejores validaciones
@app.put("/movimientos/{movimiento_id}")
def actualizar_movimiento(movimiento_id: int, movimiento: MovimientoUpdate, db:
Session = Depends(get_db)):
    """
    Endpoint unificado - redirige a actualizar_egreso si es egreso
    """
    mov = db.query(models.Movimiento).join(
        models.Descripcion,
        models.Movimiento.descripcion_id == models.Descripcion.id
    ).filter(models.Movimiento.id == movimiento_id).first()

    if not mov:
        raise HTTPException(status_code=404, detail="Movimiento no encontrado")

    # Si es egreso, usar la lógica específica de egresos
    if mov.descripcion.tipo == "egreso":
        return actualizar_egreso(movimiento_id, movimiento, db)

    # Para ingresos, lógica simple (sin recálculo de deudas)
    if movimiento.descripcion_id:
        mov.descripcion_id = movimiento.descripcion_id
    if movimiento.monto:
        mov.monto = Decimal(str(movimiento.monto))
```

```python
    if movimiento.fecha:
        mov.fecha = movimiento.fecha
    if movimiento.deuda_id is not None:
        mov.deuda_id = movimiento.deuda_id

    db.commit()
    db.refresh(mov)

    return {
        "mensaje": "Ingreso actualizado correctamente",
        "movimiento": {
            "id": mov.id,
            "descripcion_id": mov.descripcion_id,
            "tipo": mov.descripcion.tipo,
            "categoria": mov.descripcion.nombre,
            "monto": str(mov.monto),
            "fecha": mov.fecha.isoformat(),
            "deuda_id": mov.deuda_id
        }
    }

# Endpoint: devuelve el total de ingresos filtrados por rango de fechas
@app.get("/ingresos/total")
def total_ingresos(
    desde: date | None = Query(None, description="Fecha desde"),
    hasta: date | None = Query(None, description="Fecha hasta"),
    db: Session = Depends(get_db)
):
    # JOIN agregado para filtrar por tipo
    query = db.query(models.Movimiento).join(
        models.Descripcion,
        models.Movimiento.descripcion_id == models.Descripcion.id
    ).filter(models.Descripcion.tipo == "ingreso")  # Filtro cambiado

    if desde:
        query = query.filter(models.Movimiento.fecha >= desde)
    if hasta:
        query = query.filter(models.Movimiento.fecha <= hasta)

    total = query.count()
    return {"total": total}

# ✅ AJUSTE 2: Endpoint para obtener fecha mínima (auto-completar)
@app.get("/movimientos/fecha-minima")
def get_fecha_minima(db: Session = Depends(get_db)):
    """Retorna la fecha más antigua de movimientos (para auto-completar filtros)"""
    fecha_min = db.query(func.min(models.Movimiento.fecha)).scalar()
    return {
        "fecha_minima": fecha_min.strftime("%Y-%m-%d") if fecha_min else None
    }
```

```python
#app para implementar el boton anterior y siguiente de la tabla lista de ingresos
# ✅ Límite máximo de registros permitidos
MAX_RECORDS = 1000

@app.get("/ingresos/paginados")
def get_ingresos_paginados(
    page: int = Query(1, ge=1),
    limit: int = Query(4, ge=1),
    fecha_inicio: date = Query(None),
    fecha_fin: date = Query(None),
    descripcion_id: int = Query(None),  # ✅ NUEVO parámetro
    db: Session = Depends(get_db)
):
    query = db.query(models.Movimiento).join(
        models.Descripcion,
        models.Movimiento.descripcion_id == models.Descripcion.id
    ).filter(models.Descripcion.tipo == "ingreso")

    if fecha_inicio:
        query = query.filter(models.Movimiento.fecha >= fecha_inicio)
    if fecha_fin:
        query = query.filter(models.Movimiento.fecha <= fecha_fin)
    # ✅ NUEVO: Filtrar por descripción si se especifica
    if descripcion_id:
        query = query.filter(models.Movimiento.descripcion_id == descripcion_id)

    query = query.order_by(models.Movimiento.fecha.desc(),
models.Movimiento.id.desc())

    # ✅ AJUSTE 1 (TOPE 2): Validar límite de registros
    total = query.count()

    if total > MAX_RECORDS:
        raise HTTPException(
            status_code=400,
            detail=f"⚠ Demasiados resultados: {total} registros (máximo: {MAX_RECORDS}).
Use un rango de fechas más específico."
        )

    # Aplicar paginación
    ingresos = query.offset((page - 1) * limit).limit(limit).all()

    return {
        "page": page,
        "limit": limit,
        "total": total,
        "results": [ingreso.to_dict() for ingreso in ingresos]
    }

# Endpoint: eliminar un ingreso por ID
@app.delete("/ingresos/{id}")
```

```python
def delete_ingreso(id: int, db: Session = Depends(get_db)):
    # JOIN agregado para filtrar por tipo
    ingreso = db.query(models.Movimiento).join(
        models.Descripcion,
        models.Movimiento.descripcion_id == models.Descripcion.id
    ).filter(
        models.Movimiento.id == id,
        models.Descripcion.tipo == "ingreso"  # Filtro cambiado
    ).first()

    if not ingreso:
        raise HTTPException(status_code=404, detail="Ingreso no encontrado")

    db.delete(ingreso)
    db.commit()
    return {"message": "Ingreso eliminado correctamente"}

@app.get("/movimientos/categorias")
async def obtener_categorias_movimientos():
    """
    Devuelve todas las descripciones de tipos 'ingreso' y 'egreso'
    """
    try:
        with engine.connect() as conn:
            query = text("""
                SELECT DISTINCT nombre
                FROM descripciones
                WHERE tipo IN ('ingreso', 'egreso') AND activa = 1
                ORDER BY nombre ASC
            """)

            resultado = conn.execute(query)
            categorias = [fila[0] for fila in resultado]

            return categorias

    except Exception as e:
        print(f"Error obteniendo categorías de movimientos: {e}")
        raise HTTPException(status_code=500, detail="Error al obtener categorías")

## Prueba de los endpoints:
# REEMPLAZAR COMPLETAMENTE el endpoint POST /egresos:
"""
@app.post("/egresos")
def crear_egreso(movimiento: MovimientoCreate, db: Session = Depends(get_db)):

    # Crear egreso con lógica específica integrada (no depende de crear_movimiento)
    # CAMBIO: Antes llamaba a crear_movimiento(), ahora tiene lógica propia
    # MOTIVO: Mantener consistencia con PUT /egresos que también es específico


```

```
    # PASO 1: Validaciones iniciales
    # Convertir monto a Decimal para precisión monetaria
    monto_pago = Decimal(str(movimiento.monto))

    # Convertir deuda_id a int o None
    deuda_id = int(movimiento.deuda_id) if movimiento.deuda_id else None

    # PASO 2: Validar que la descripción sea tipo "egreso"
    # EXPLICACIÓN: Esto previene crear un "egreso" con descripción de "ingreso"
    descripcion = db.query(models.Descripcion).filter(
        models.Descripcion.id == movimiento.descripcion_id
    ).first()

    if not descripcion:
        raise HTTPException(status_code=400, detail="Descripción no encontrada")

    if descripcion.tipo != "egreso":
        raise HTTPException(
            status_code=400,
            detail=f"La descripción seleccionada es tipo '{descripcion.tipo}', debe ser
'egreso'"
        )

    # PASO 3: Crear el movimiento (egreso)
    nuevo_egreso = models.Movimiento(
        descripcion_id=movimiento.descripcion_id,
        monto=monto_pago,
        fecha=movimiento.fecha or date.today(),
        deuda_id=deuda_id
    )
    db.add(nuevo_egreso)

    # PASO 4: Actualizar deuda si corresponde (lógica de pago)
    # EXPLICACIÓN: Si el egreso paga una deuda, actualizamos los campos financieros
    if deuda_id:
        deuda = db.query(models.Deuda).filter(models.Deuda.id == deuda_id).first()

        if not deuda:
            raise HTTPException(status_code=400, detail="Deuda no encontrada")

        if deuda:
            # Obtener valores actuales
            monto_deuda = Decimal(str(deuda.monto))
            monto_pagado_actual = Decimal(str(deuda.monto_pagado or '0.00'))

            # Sumar el nuevo pago
            deuda.monto_pagado = (monto_pagado_actual +
monto_pago).quantize(Decimal('0.01'))

            # Recalcular saldo pendiente
```

```
        deuda.saldo_pendiente = (monto_deuda -
deuda.monto_pagado).quantize(Decimal('0.01'))

        # Actualizar estado de pagado
        if deuda.saldo_pendiente <= Decimal('0.00'):
            deuda.pagado = True
            deuda.saldo_pendiente = Decimal('0.00')  # Asegurar que quede en 0

    # PASO 5: Confirmar cambios
    db.commit()
    db.refresh(nuevo_egreso)

    # PASO 6: Preparar respuesta
    # EXPLICACIÓN: Usamos la relación cargada para obtener el nombre
    descripcion_nombre = nuevo_egreso.descripcion.nombre if nuevo_egreso.descripcion
else ""

    return {
        "mensaje": "Egreso registrado correctamente",
        "movimiento": {
            "id": nuevo_egreso.id,
            "descripcion_id": nuevo_egreso.descripcion_id,
            "categoria": descripcion_nombre,
            "monto": str(nuevo_egreso.monto),
            "fecha": nuevo_egreso.fecha.isoformat(),
            "deuda_id": nuevo_egreso.deuda_id
        }
    }
"""
# IMPORTANTE: Después de este cambio, puedes ELIMINAR o COMENTAR:
# - @app.post("/movimientos") y su función crear_movimiento()
# - O dejarlo solo para ingresos si lo usas ahí

# PERO: Si ingresos usa POST /ingresos que llama a crear_movimiento(),
# deberás hacer el mismo cambio para ingresos en el Paso 2


@app.post("/egresos")
def crear_egreso(movimiento: MovimientoCreate, db: Session = Depends(get_db)):
    try:
        # 1. Validar descripción tipo "egreso"
        descripcion = db.query(models.Descripcion).filter(
            models.Descripcion.id == movimiento.descripcion_id
        ).first()
        if not descripcion:
            raise HTTPException(status_code=400, detail="Descripción no encontrada")
        if descripcion.tipo != "egreso":
            raise HTTPException(
                status_code=400,
                detail=f"La descripción debe ser tipo 'egreso', no '{descripcion.tipo}'"
            )
```

```
    # 2. Convertir monto a Decimal (soporta "1.500,50", "1500.50", "1500")
    try:
        monto_str = str(movimiento.monto).strip()
        # Eliminar puntos de miles, reemplazar coma por punto
        monto_str = monto_str.replace('.', '', monto_str.count('.') - 1).replace(',',
'.')
        monto_pago = Decimal(monto_str).quantize(Decimal('0.01'))
        if monto_pago <= Decimal('0.00'):
            raise HTTPException(status_code=400, detail="El monto debe ser mayor a
$0,00")
    except (InvalidOperation, ValueError, TypeError) as e:
        raise HTTPException(status_code=400, detail=f"Monto inválido:
'{movimiento.monto}'")

    # 3. Manejo de deuda (si aplica)
    deuda_saldo_antes = None  # ← INICIALIZAR UNA SOLA VEZ
    deuda_id = None

    if movimiento.deuda_id:
        deuda_id = int(movimiento.deuda_id)
        deuda = db.query(models.Deuda).filter(models.Deuda.id == deuda_id).first()
        if not deuda:
            raise HTTPException(status_code=400, detail="Deuda no encontrada")
        if deuda.saldo_pendiente <= Decimal('0.00'):
            raise HTTPException(status_code=400, detail="Deuda ya está pagada")
        if monto_pago > deuda.saldo_pendiente:
            raise HTTPException(
                status_code=400,
                detail=f"El monto supera el saldo pendiente (${deuda.saldo_pendiente})"
            )
        deuda_saldo_antes = deuda.saldo_pendiente  # ← CAPTURAR SALDO ANTES

    # 4. Crear movimiento
    # ← ELIMINAR la línea "deuda_saldo_antes = None" que estaba aquí
    nuevo_egreso = models.Movimiento(
        descripcion_id=movimiento.descripcion_id,
        monto=monto_pago,
        fecha=movimiento.fecha or date.today(),
        deuda_id=deuda_id,
        deuda_saldo_antes=deuda_saldo_antes  # ← USAR el valor capturado antes
    )
    db.add(nuevo_egreso)

    # 5. Actualizar deuda (si aplica)
    if deuda_id and deuda_saldo_antes is not None:
        deuda.monto_pagado = (deuda.monto_pagado or Decimal('0.00')) + monto_pago
        deuda.saldo_pendiente = deuda.saldo_pendiente - monto_pago
        if deuda.saldo_pendiente <= Decimal('0.00'):
            deuda.saldo_pendiente = Decimal('0.00')
            deuda.pagado = True  # ← CORREGIDO: era "esta_pagada", ahora "pagado"
```

```
    # 6. Confirmar
    db.commit()
    db.refresh(nuevo_egreso)

    # 7. Responder
    return {
      "mensaje": "Egreso registrado correctamente",
      "movimiento": {
        "id": nuevo_egreso.id,
        "descripcion_id": nuevo_egreso.descripcion_id,
        "categoria": descripcion.nombre,
        "monto": str(nuevo_egreso.monto),
        "fecha": nuevo_egreso.fecha.isoformat(),
        "deuda_id": nuevo_egreso.deuda_id,
        "deuda_saldo_antes": str(nuevo_egreso.deuda_saldo_antes) if
nuevo_egreso.deuda_saldo_antes else None
      }
    }

  except HTTPException:
    raise
  except Exception as e:
    db.rollback()
    raise HTTPException(status_code=500, detail=f"Error interno: {str(e)}")
# Endpoint para obtener egresos paginados y filtrados por fecha
"""
@app.get("/egresos/paginados")
def obtener_egresos_paginados(
  page: int = Query(1, ge=1),
  limit: int = Query(4, ge=1),
  fecha_inicio: Optional[str] = Query(None),
  fecha_fin: Optional[str] = Query(None),
  descripcion_id: Optional[int] = Query(None),
  db: Session = Depends(get_db)
):
  # ✅ JOIN con Descripcion + LEFT JOIN con Deuda
  query = (
    db.query(models.Movimiento, models.Descripcion, models.Deuda)
    .join(models.Descripcion, models.Movimiento.descripcion_id ==
models.Descripcion.id)
    .outerjoin(models.Deuda, models.Movimiento.deuda_id == models.Deuda.id)
    .filter(models.Descripcion.tipo == "egreso")
  )

  # Filtros por fecha
  if fecha_inicio:
    try:
      fecha_desde = datetime.strptime(fecha_inicio, "%Y-%m-%d").date()
      query = query.filter(models.Movimiento.fecha >= fecha_desde)
    except ValueError:
```

```
            raise HTTPException(status_code=400, detail="Formato de fecha_inicio
inválido, debe ser YYYY-MM-DD")

    if fecha_fin:
        try:
            fecha_hasta = datetime.strptime(fecha_fin, "%Y-%m-%d").date()
            query = query.filter(models.Movimiento.fecha <= fecha_hasta)
        except ValueError:
            raise HTTPException(status_code=400, detail="Formato de fecha_fin inválido,
debe ser YYYY-MM-DD")

    # Filtro por descripción
    if descripcion_id is not None:
        query = query.filter(models.Movimiento.descripcion_id == descripcion_id)

    # Ordenar: más recientes primero
    query = query.order_by(models.Movimiento.fecha.desc(),
models.Movimiento.id.desc())

    # Paginación
    total = query.count()
    results = query.offset((page - 1) * limit).limit(limit).all()

    # ✅ Construir resultado seguro: SOLO PRIMITIVOS (str, float, int, None)
    egresos_list = []
    for movimiento, descripcion, deuda in results:
        item = {
            "id": movimiento.id,
            "descripcion_id": movimiento.descripcion_id,
            "categoria": getattr(descripcion, 'nombre', 'Sin descripción'),
            "monto": float(movimiento.monto) if movimiento.monto else 0.0,
            "fecha": movimiento.fecha.isoformat() if movimiento.fecha else None,
            "deuda_id": None,
            "deuda_descripcion": None,
            "deuda_saldo_pendiente": None
        }

        # ✅ Solo si hay deuda (y no es None)
        if deuda is not None:
            item["deuda_id"] = deuda.id
            item["deuda_descripcion"] = (
                getattr(deuda, 'descripcion', None) or
                getattr(deuda, 'nombre', None) or
                'Deuda sin nombre'
            )
            saldo = getattr(deuda, 'saldo_pendiente', None) or getattr(deuda, 'monto',
0.0)
            item["deuda_saldo_pendiente"] = float(saldo) if saldo is not None else 0.0

        egresos_list.append(item)
```

```
    return{
        "page": page,
        "limit": limit,
        "total": total,
        "results": egresos_list
    }
"""

# REEMPLAZAR COMPLETAMENTE el endpoint GET /egresos/paginados:
@app.get("/egresos/paginados")
def obtener_egresos_paginados(
    page: int = Query(1, ge=1),
    limit: int = Query(4, ge=1),
    fecha_inicio: Optional[str] = Query(None),
    fecha_fin: Optional[str] = Query(None),
    descripcion_id: Optional[int] = Query(None),
    db: Session = Depends(get_db)
):
    # ✅ JOIN con Descripcion + LEFT JOIN con Deuda
    query = (
        db.query(models.Movimiento, models.Descripcion, models.Deuda)
        .join(models.Descripcion, models.Movimiento.descripcion_id == 
models.Descripcion.id)
        .outerjoin(models.Deuda, models.Movimiento.deuda_id == models.Deuda.id)
        .filter(models.Descripcion.tipo == "egreso")
    )

    # Filtros por fecha
    if fecha_inicio:
        try:
            fecha_desde = datetime.strptime(fecha_inicio, "%Y-%m-%d").date()
            query = query.filter(models.Movimiento.fecha >= fecha_desde)
        except ValueError:
            raise HTTPException(status_code=400, detail="Formato de fecha_inicio 
inválido, debe ser YYYY-MM-DD")

    if fecha_fin:
        try:
            fecha_hasta = datetime.strptime(fecha_fin, "%Y-%m-%d").date()
            query = query.filter(models.Movimiento.fecha <= fecha_hasta)
        except ValueError:
            raise HTTPException(status_code=400, detail="Formato de fecha_fin inválido, 
debe ser YYYY-MM-DD")

    # Filtro por descripción
    if descripcion_id is not None:
        query = query.filter(models.Movimiento.descripcion_id == descripcion_id)

    # Ordenar: más recientes primero
    query = query.order_by(models.Movimiento.fecha.desc(), 
models.Movimiento.id.desc())
```

```
  # Paginación
  total = query.count()
  results = query.offset((page - 1) * limit).limit(limit).all()

  # ✅ Construir resultado seguro: solo primitivos (str, float, int, None)
  egresos_list = []
  for movimiento, descripcion, deuda in results:
    item = {
      "id": movimiento.id,
      "descripcion_id": movimiento.descripcion_id,
      "categoria": getattr(descripcion, 'nombre', 'Sin descripción'),
      "monto": float(movimiento.monto) if movimiento.monto else 0.0,
      "fecha": movimiento.fecha.isoformat() if movimiento.fecha else None,
      "deuda_id": movimiento.deuda_id,
      "deuda_descripcion": None,
      "deuda_saldo_pendiente": None,
      "deuda_saldo_antes": float(movimiento.deuda_saldo_antes) if
movimiento.deuda_saldo_antes is not None else None,
    }

    # ✅ Solo si hay deuda asociada
    if deuda is not None:
      item["deuda_descripcion"] = (
        getattr(deuda, 'descripcion', None) or
        getattr(deuda, 'nombre', None) or
        'Deuda sin nombre'
      )
      saldo = getattr(deuda, 'saldo_pendiente', None) or getattr(deuda, 'monto',
0.0)
      item["deuda_saldo_pendiente"] = float(saldo) if saldo is not None else 0.0

    egresos_list.append(item)

  return {
    "page": page,
    "limit": limit,
    "total": total,
    "results": egresos_list
  }

# Endpoint: devuelve el total de egresos filtrados por rango de fechas
@app.get("/egresos/total")
def total_egresos(
  desde: date | None = Query(None, description="Fecha desde"),
  hasta: date | None = Query(None, description="Fecha hasta"),
  db: Session = Depends(get_db)
):
  # JOIN agregado para filtrar por tipo
  query = db.query(models.Movimiento).join(
    models.Descripcion,
```

```
    models.Movimiento.descripcion_id == models.Descripcion.id
  ).filter(models.Descripcion.tipo == "egreso")  # Filtro cambiado

  if desde:
    query = query.filter(models.Movimiento.fecha >= desde)
  if hasta:
    query = query.filter(models.Movimiento.fecha <= hasta)

  total = query.count()
  return {"total": total}

# PASO 1: CAMBIOS EN main.py - ENDPOINTS ESPECÍFICOS PARA EGRESOS

# 1. REEMPLAZAR COMPLETAMENTE este endpoint:
@app.put("/egresos/{egreso_id}")
def actualizar_egreso(egreso_id: int, movimiento: MovimientoUpdate, db: Session =
Depends(get_db)):
  """
  Endpoint específico para actualizar egresos con lógica de recálculo de deuda

  CAMBIO: Antes llamaba a actualizar_movimiento(), ahora tiene lógica propia
  MOTIVO: Los egresos necesitan recálculo automático de deuda, los ingresos no
  """

  # PASO 1: Buscar el egreso con JOIN para validar que existe y es un egreso
  # EXPLICACIÓN: Usamos JOIN para asegurar que el movimiento sea tipo "egreso"
  # y cargar la información de descripción en una sola consulta
  egreso = db.query(models.Movimiento).join(
    models.Descripcion,
    models.Movimiento.descripcion_id == models.Descripcion.id
  ).filter(
    models.Movimiento.id == egreso_id,
    models.Descripcion.tipo == "egreso"  # FILTRO: Solo egresos
  ).first()

  # VALIDACIÓN: Si no existe o no es egreso, error 404
  if not egreso:
    raise HTTPException(status_code=404, detail="Egreso no encontrado")

  # PASO 2: Guardar valores originales para poder hacer el recálculo correcto
  # EXPLICACIÓN: Necesitamos estos valores para revertir el pago anterior
  # antes de aplicar el nuevo monto
  monto_original = egreso.monto
  deuda_id_original = egreso.deuda_id

  # PASO 3: Validación de negocio - No cambiar descripción si tiene deuda
  # EXPLICACIÓN: Cambiar la descripción de un egreso que pagó una deuda
  # podría causar inconsistencias contables
  if egreso.deuda_id and movimiento.descripcion_id and movimiento.descripcion_id !=
egreso.descripcion_id:
    raise HTTPException(
```

```
            status_code=400,
            detail="No se puede cambiar la descripción de un egreso vinculado a deuda"
        )

    # PASO 4: Actualizar campos del egreso
    # EXPLICACIÓN: Solo actualizamos los campos que vienen en el request
    # Si un campo es None, no lo tocamos (mantiene el valor actual)
    if movimiento.descripcion_id:
        egreso.descripcion_id = movimiento.descripcion_id
    if movimiento.monto:
        # IMPORTANTE: Convertir a Decimal para precisión monetaria
        egreso.monto = Decimal(str(movimiento.monto))
    if movimiento.fecha:
        egreso.fecha = movimiento.fecha
    # NOTA: deuda_id NO se actualiza - está bloqueado en el frontend

    # PASO 5: LÓGICA DE RECÁLCULO DE DEUDA (la parte más importante)
    # EXPLICACIÓN: Solo recalculamos si cambió el monto Y tiene deuda vinculada
    if egreso.deuda_id and egreso.monto != monto_original:
        # Buscar la deuda vinculada
        deuda = db.query(models.Deuda).filter(models.Deuda.id ==
egreso.deuda_id).first()

        if deuda:
            # PASO 5.1: Revertir el pago original
            # EXPLICACIÓN: Quitamos el monto anterior del total pagado
            deuda.monto_pagado = (deuda.monto_pagado -
monto_original).quantize(Decimal('0.01'))

            # PASO 5.2: Aplicar el nuevo pago
            # EXPLICACIÓN: Sumamos el nuevo monto al total pagado
            deuda.monto_pagado = (deuda.monto_pagado +
egreso.monto).quantize(Decimal('0.01'))

            # PASO 5.3: Recalcular saldo pendiente
            # EXPLICACIÓN: saldo_pendiente = monto_total - monto_pagado
            monto_deuda = Decimal(str(deuda.monto))
            deuda.saldo_pendiente = (monto_deuda -
deuda.monto_pagado).quantize(Decimal('0.01'))

            # PASO 5.4: Actualizar estado de pagado
            # EXPLICACIÓN: Si saldo_pendiente <= 0, la deuda está totalmente pagada
            if deuda.saldo_pendiente <= Decimal('0.00'):
                deuda.pagado = True
                deuda.saldo_pendiente = Decimal('0.00')  # Asegurar que quede en 0
            else:
                deuda.pagado = False  # Vuelve a pendiente si quedó saldo

    # PASO 6: Confirmar cambios en base de datos
    db.commit()
    db.refresh(egreso)  # Recargar datos actualizados
```

```
    # PASO 7: Preparar respuesta
    # EXPLICACIÓN: Devolvemos los datos actualizados en el formato esperado
    return {
        "mensaje": "Egreso actualizado correctamente",
        "movimiento": {
            "id": egreso.id,
            "descripcion_id": egreso.descripcion_id,
            "categoria": egreso.descripcion.nombre,  # Nombre de la descripción
            "monto": str(egreso.monto),
            "fecha": egreso.fecha.isoformat(),
            "deuda_id": egreso.deuda_id
        }
    }


"""
# Endpoint: eliminar un egreso por ID
@app.delete("/egresos/{id}")
def delete_egreso(id: int, db: Session = Depends(get_db)):
    # JOIN agregado para filtrar por tipo
    egreso = db.query(models.Movimiento).join(
        models.Descripcion,
        models.Movimiento.descripcion_id == models.Descripcion.id
    ).filter(
        models.Movimiento.id == id,
        models.Descripcion.tipo == "egreso"  # Filtro cambiado
    ).first()

    if not egreso:
        raise HTTPException(status_code=404, detail="Egreso no encontrado")

    db.delete(egreso)
    db.commit()
    return {"message": "Egreso eliminado correctamente"}
"""

# 2. MANTENER IGUAL - Endpoint de eliminación con reversión:
@app.delete("/egresos/{id}")
def delete_egreso(id: int, db: Session = Depends(get_db)):
    """
    Eliminar egreso con reversión automática de deuda

    CAMBIO: Agregamos lógica de reversión automática
    MOTIVO: Al eliminar un egreso que pagó una deuda, debemos revertir el pago
    """

    # PASO 1: Buscar el egreso a eliminar
    egreso = db.query(models.Movimiento).join(
        models.Descripcion,
        models.Movimiento.descripcion_id == models.Descripcion.id
```

```
    ).filter(
        models.Movimiento.id == id,
        models.Descripcion.tipo == "egreso"
    ).first()

    if not egreso:
        raise HTTPException(status_code=404, detail="Egreso no encontrado")

    # PASO 2: REVERSIÓN AUTOMÁTICA - La lógica clave
    # EXPLICACIÓN: Si el egreso que vamos a eliminar pagó una deuda,
    # debemos revertir ese pago para mantener consistencia
    if egreso.deuda_id:
        deuda = db.query(models.Deuda).filter(models.Deuda.id ==
egreso.deuda_id).first()

        if deuda:
            # REVERSIÓN: Quitar el pago del egreso eliminado
            monto_egreso = egreso.monto
            deuda.monto_pagado = (deuda.monto_pagado -
monto_egreso).quantize(Decimal('0.01'))

            # Recalcular saldo pendiente
            monto_deuda = Decimal(str(deuda.monto))
            deuda.saldo_pendiente = (monto_deuda -
deuda.monto_pagado).quantize(Decimal('0.01'))

            # Actualizar estado - Si queda saldo, vuelve a pendiente
            if deuda.saldo_pendiente > Decimal('0.00'):
                deuda.pagado = False

    # PASO 3: Eliminar el egreso
    db.delete(egreso)
    db.commit()

    return {"message": "Egreso eliminado correctamente y deuda actualizada"}

# 3. MANTENER IGUAL - Los endpoints POST no cambian
# EXPLICACIÓN: La lógica de creación ya funciona correctamente
# No necesitamos tocarla

@app.get("/deudas")
def obtener_deudas(db: Session = Depends(get_db)):
    resultados = db.query(models.Deuda).all()
    return [d.to_dict() for d in resultados]

@app.post("/deudas")
def crear_deuda(deuda: DeudaCreate, db: Session = Depends(get_db)):
    nueva_deuda = models.Deuda(
        # Eliminado: descripcion=deuda.descripcion,
        descripcion_id=deuda.descripcion_id,  # NUEVO
        monto=deuda.monto,
```

```
        fecha_registro=deuda.fecha_registro or date.today(),
        fecha_vencimiento=deuda.fecha_vencimiento,
        pagado=0,
        # CORRECCIÓN: Agregar estas líneas que faltaban
        monto_pagado=Decimal('0.00'),
        saldo_pendiente=Decimal(str(deuda.monto))  # Inicializar con el monto total
    )
    db.add(nueva_deuda)
    db.commit()
    db.refresh(nueva_deuda)

    return {
        "mensaje": "Deuda registrada correctamente",
        "deuda": nueva_deuda.to_dict()
    }


@app.put("/deudas/{deuda_id}/pagar")
def pagar_deuda(deuda_id: int, db: Session = Depends(get_db)):
    deuda = db.query(models.Deuda).filter(models.Deuda.id == deuda_id).first()
    if not deuda:
        return JSONResponse(status_code=404, content={"mensaje": "Deuda no
encontrada"})

    deuda.pagado = 1
    db.commit()
    db.refresh(deuda)

    return {"mensaje": "Deuda marcada como pagada", "deuda": deuda.to_dict()}

# ALTERNATIVA MÁS SIMPLE: Modificar el endpoint existente en main.py
# Deudas no pagadas (línea ~xxx)
@app.get("/deudas/no_pagadas")
def obtener_deudas_no_pagadas(
    desde: date = Query(None),
    hasta: date = Query(None),
    include_deuda_id: int = Query(None),
    descripcion: str = Query(None),  # ← AGREGAR
    db: Session = Depends(get_db)
):
    query = db.query(models.Deuda).join(
        models.Descripcion,
        models.Deuda.descripcion_id == models.Descripcion.id
    ).filter(models.Deuda.pagado == 0)

    if desde:
        query = query.filter(models.Deuda.fecha_registro >= desde)
    if hasta:
        query = query.filter(models.Deuda.fecha_registro <= hasta)

    # ← NUEVO
```

```
    if descripcion:
        query = query.filter(models.Descripcion.nombre == descripcion)

    deudas = query.all()

    if include_deuda_id:
        deuda_especifica = db.query(models.Deuda).filter(
            models.Deuda.id == include_deuda_id
        ).first()
        if deuda_especifica:
            ids_existentes = [d.id for d in deudas]
            if deuda_especifica.id not in ids_existentes:
                deudas.insert(0, deuda_especifica)

    return [d.to_dict() for d in deudas]

@app.get("/deudas/pagadas")
def obtener_deudas_pagadas(
    desde: date | None = Query(None),
    hasta: date | None = Query(None),
    descripcion: str = Query(None),  # ← AGREGAR
    db: Session = Depends(get_db)
):
    query = db.query(models.Deuda).filter(models.Deuda.pagado == 1)

    if desde:
        query = query.filter(models.Deuda.fecha_registro >= desde)
    if hasta:
        query = query.filter(models.Deuda.fecha_registro <= hasta)

    # ← NUEVO
    if descripcion:
        query = query.filter(models.Descripcion.nombre == descripcion)

    deudas = query.order_by(models.Deuda.fecha_registro.desc()).all()
    return [d.to_dict() for d in deudas]

# Endpoint: devuelve todas las deudas (pagadas y no pagadas), con filtros
opcionales por fecha
@app.get("/deudas/todas")
def obtener_deudas_todas(
    desde: date = Query(None),
    hasta: date = Query(None),
    descripcion: str = Query(None),  # ← AGREGAR
    db: Session = Depends(get_db)
):
    query = db.query(models.Deuda)

    if desde:
        query = query.filter(models.Deuda.fecha_registro >= desde)
    if hasta:
```

```
    query = query.filter(models.Deuda.fecha_registro <= hasta)

  # ← NUEVO
  if descripcion:
    query = query.filter(models.Descripcion.nombre == descripcion)

  # Ordeno por fecha_registro para que la tabla salga en orden cronológico
  deudas = query.order_by(models.Deuda.fecha_registro).all()
  return [d.to_dict() for d in deudas]

@app.get("/deudas/paginadas")
def get_deudas_paginadas(
  page: int = Query(1, ge=1),
  limit: int = Query(4, ge=1),
  fecha_inicio: date = Query(None),
  fecha_fin: date = Query(None),
  estado: str = None,
  descripcion_id: int = Query(None),
  db: Session = Depends(get_db)
):
  # CRÍTICO: Expirar cache de la sesión antes de consultar
  # EXPLICACIÓN: SQLAlchemy cachea objetos en la sesión. Si modificaste
  # deudas desde otro endpoint (egresos), la sesión puede tener datos viejos.
  # expire_all() fuerza a SQLAlchemy a recargar TODOS los datos desde la BD
  db.expire_all()

  # Query base con JOIN explícito
  query = db.query(models.Deuda).join(
    models.Descripcion,
    models.Deuda.descripcion_id == models.Descripcion.id
  )

  # ✅ AUTO-COMPLETAR FECHAS SI NO VIENEN (búsqueda total)
  if fecha_inicio is None and fecha_fin is None:
    # Buscar la fecha más antigua en la BD
    fecha_min_obj = db.query(func.min(models.Deuda.fecha_registro)).scalar()
    if fecha_min_obj:
      fecha_inicio = fecha_min_obj
    # Fecha fin = HOY
    fecha_fin = date.today()

  # Aplicar filtro de fechas
  if fecha_inicio:
    try:
      query = query.filter(models.Deuda.fecha_registro >= fecha_inicio)
    except ValueError:
      return {"error": "Formato de fecha_inicio inválido, debe ser YYYY-MM-DD"}
  if fecha_fin:
    try:
      query = query.filter(models.Deuda.fecha_registro <= fecha_fin)
    except ValueError:
```

```
            return {"error": "Formato de fecha_fin inválido, debe ser YYYY-MM-DD"}

    # Filtro por descripción
    if descripcion_id:
        query = query.filter(models.Deuda.descripcion_id == descripcion_id)

    # Filtro por estado
    if estado == "pagados":
        query = query.filter(models.Deuda.pagado == True)
    elif estado == "pendientes":
        query = query.filter(models.Deuda.pagado == False)

    # Ordenar por fecha_registro DESC y por id DESC (más recientes primero)
    query = query.order_by(models.Deuda.fecha_registro.desc(),
models.Deuda.id.desc())

    # Aplicar paginación
    total = query.count()
    deudas = query.offset((page - 1) * limit).limit(limit).all()

    return {
        "page": page,
        "limit": limit,
        "total": total,
        "results": [d.to_dict() for d in deudas]
    }

@app.delete("/deudas/{id}")
def delete_deuda(id: int, db: Session = Depends(get_db)):
    # Buscar la deuda con JOIN para consistencia
    deuda = db.query(models.Deuda).join(
        models.Descripcion,
        models.Deuda.descripcion_id == models.Descripcion.id
    ).filter(models.Deuda.id == id).first()

    if not deuda:
        raise HTTPException(status_code=404, detail="Deuda no encontrada")

    # Verificar si tiene egresos (pagos) asociados
    egresos_asociados = db.query(models.Movimiento).filter(
        models.Movimiento.deuda_id == id
    ).count()

    if egresos_asociados > 0:
        raise HTTPException(
            status_code=400,
            detail=f"No se puede eliminar la deuda. Tiene {egresos_asociados} egreso(s)
asociado(s)"
        )

    db.delete(deuda)
```

```
    db.commit()
    return {"message": "Deuda eliminada correctamente"}

@app.put("/deudas/{deuda_id}")
def actualizar_deuda(deuda_id: int, deuda_data: DeudaUpdate, db: Session =
Depends(get_db)):
    # Buscar deuda con JOIN para consistencia
    deuda = db.query(models.Deuda).join(
        models.Descripcion,
        models.Deuda.descripcion_id == models.Descripcion.id
    ).filter(models.Deuda.id == deuda_id).first()

    if not deuda:
        raise HTTPException(status_code=404, detail="Deuda no encontrada")

    # Validar descripcion_id si se está actualizando
    if deuda_data.descripcion_id is not None:
        descripcion_existe = db.query(models.Descripcion).filter(
            models.Descripcion.id == deuda_data.descripcion_id
        ).first()
        if not descripcion_existe:
            raise HTTPException(status_code=400, detail="Descripción no encontrada")
        deuda.descripcion_id = deuda_data.descripcion_id

    # Validar monto si se está actualizando
    if deuda_data.monto is not None:
        if deuda_data.monto <= 0:
            raise HTTPException(status_code=400, detail="El monto debe ser mayor a 0")
        deuda.monto = deuda_data.monto

        # ✅ AJUSTE 3: Recalcular saldo cuando se edita el monto
        # EXPLICACIÓN: Si cambia el monto, debemos recalcular el saldo
        # Fórmula: saldo_pendiente = nuevo_monto - monto_pagado
        monto_pagado = deuda.monto_pagado or 0
        deuda.saldo_pendiente = deuda_data.monto - monto_pagado

    if deuda_data.fecha_registro is not None:
        deuda.fecha_registro = deuda_data.fecha_registro
    if deuda_data.fecha_vencimiento is not None:
        deuda.fecha_vencimiento = deuda_data.fecha_vencimiento

    db.commit()
    db.refresh(deuda)

    return {
        "mensaje": "Deuda actualizada correctamente",
        "deuda": deuda.to_dict()
    }

@app.get("/deudas/fecha-minima")
def get_fecha_minima_deudas(db: Session = Depends(get_db)):
```

```
    """
    Retorna la fecha más antigua registrada en la tabla deudas.
    Usado para auto-completar filtros de búsqueda total.
    """
    fecha_min = db.query(func.min(models.Deuda.fecha_registro)).scalar()

    return {
        "fecha_minima": fecha_min.isoformat() if fecha_min else None
    }

@app.get("/deudas/descripciones")
async def obtener_descripciones_deudas():
    """
    Devuelve todas las descripciones de tipo 'deuda'
    """
    try:
        # Conectar usando el engine de SQLAlchemy
        with engine.connect() as conn:
            # text() envuelve la consulta SQL para que SQLAlchemy la entienda
            query = text("""
                SELECT DISTINCT nombre
                FROM descripciones
                WHERE tipo = 'deuda' AND activa = 1
                ORDER BY nombre ASC
            """)

            # Ejecutar la consulta
            resultado = conn.execute(query)

            # Extraer solo los nombres (cada fila es un objeto Row)
            descripciones = [fila[0] for fila in resultado]

            return descripciones

    except Exception as e:
        print(f"Error obteniendo descripciones de deudas: {e}")
        raise HTTPException(status_code=500, detail="Error al obtener descripciones")


# 📌 Endpoint para obtener descripciones filtradas por tipo
@app.get("/descripciones/tipo/{tipo}")
def get_descripciones_por_tipo(
    tipo: str,
    solo_nombres: bool = Query(False),  # ← NUEVO PARÁMETRO
    db: Session = Depends(get_db)
):
    """
    Obtiene descripciones de un tipo específico
    - Si solo_nombres=true: devuelve lista de strings
    - Si solo_nombres=false: devuelve lista de objetos completos
    """
```

```
    tipos_validos = ["ingreso", "egreso", "deuda"]
    if tipo not in tipos_validos:
        raise HTTPException(
            status_code=400,
            detail=f"Tipo inválido. Debe ser uno de: {', '.join(tipos_validos)}"
        )

    descripciones = db.query(models.Descripcion).filter(
        models.Descripcion.tipo == tipo,
        models.Descripcion.activa == True
    ).order_by(models.Descripcion.nombre).all()

    # ✅ NUEVA LÓGICA: Devolver solo nombres si se solicita
    if solo_nombres:
        return [desc.nombre for desc in descripciones]  # ← Solo strings
    else:
        return [desc.to_dict() for desc in descripciones]  # ← Objetos completos

#PAGINA DE DESCRIPCIONES

# 1. CREAR UNA NUEVA DESCRIPCIÓN
@app.post("/descripciones", status_code=201)
def crear_descripcion(descripcion: DescripcionCreate, db: Session =
Depends(get_db)):
    """
    Crea una nueva descripción en la base de datos.

    Args:
        descripcion: Datos de la descripción a crear
        db: Sesión de base de datos

    Returns:
        Dict con la descripción creada
    """
    try:
        # Crear instancia del modelo
        nueva_descripcion = models.Descripcion(
            nombre=descripcion.nombre,
            tipo=descripcion.tipo,
            activa=descripcion.activa if descripcion.activa is not None else True,
            telefono=descripcion.telefono,
            email=descripcion.email,
            direccion=descripcion.direccion,
            tipo_entidad=descripcion.tipo_entidad,
            fecha_creacion=descripcion.fecha_creacion
        )

        # Guardar en la base de datos
        db.add(nueva_descripcion)
        db.commit()
        db.refresh(nueva_descripcion)
```

```
        return nueva_descripcion.to_dict()

    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al crear descripción:
{str(e)}")


# ==========================================
# ENDPOINTS DE DESCRIPCIONES CON CONTADOR DE USOS
# ==========================================

# REEMPLAZAR el endpoint GET /descripciones/paginadas actual por este:

@app.get("/descripciones/paginadas")
def obtener_descripciones_paginadas(
    page: int = 1,
    limit: int = 4,
    tipo: Optional[str] = None,
    activa: Optional[str] = None,
    nombre: Optional[str] = None,  # ← AGREGAR ESTE PARÁMETRO
    db: Session = Depends(get_db)
):
    try:
        # PASO 1: Construir query base
        query = db.query(models.Descripcion)

        # PASO 2: Aplicar filtros si existen
        if tipo:
            query = query.filter(models.Descripcion.tipo == tipo)

        if activa is not None and activa != "":
            activa_bool = activa.lower() == "true"
            query = query.filter(models.Descripcion.activa == activa_bool)

        # NUEVO: Filtro por nombre (búsqueda parcial)
        if nombre:
            query = query.filter(models.Descripcion.nombre.ilike(f"%{nombre}%"))

        # NUEVO: Ordenar por ID descendente (más recientes primero)
        query = query.order_by(models.Descripcion.id.desc())

        # PASO 3: Contar total de registros que cumplen los filtros
        total = query.count()

        # PASO 4: Calcular offset y obtener registros paginados
        offset = (page - 1) * limit
        descripciones = query.offset(offset).limit(limit).all()

        # PASO 5: NUEVO - Enriquecer cada descripción con contador de usos
```

```
    # EXPLICACIÓN: Para cada descripción, contamos cuántos movimientos
    # y deudas están vinculados a ella
    results = []
    for desc in descripciones:
        # Contar movimientos asociados (ingresos + egresos)
        movimientos_count = db.query(models.Movimiento).filter(
            models.Movimiento.descripcion_id == desc.id
        ).count()

        # Contar deudas asociadas
        deudas_count = db.query(models.Deuda).filter(
            models.Deuda.descripcion_id == desc.id
        ).count()

        # Sumar ambos contadores
        uso_total = movimientos_count + deudas_count

        # Determinar si se puede editar/eliminar
        # REGLA: Si uso_total > 0, NO se puede editar ni eliminar
        puede_modificar = uso_total == 0

        # Crear diccionario con datos enriquecidos
        desc_dict = desc.to_dict()
        desc_dict['uso_count'] = uso_total
        desc_dict['puede_editar'] = puede_modificar
        desc_dict['puede_eliminar'] = puede_modificar

        results.append(desc_dict)

    # PASO 6: Calcular total de páginas
    total_pages = (total + limit - 1) // limit if total > 0 else 1

    return {
        "results": results,
        "total": total,
        "page": page,
        "limit": limit,
        "total_pages": total_pages
    }

  except Exception as e:
    raise HTTPException(status_code=500, detail=f"Error al obtener descripciones:
{str(e)}")


# MANTENER IGUAL: Los endpoints PUT y DELETE ya tienen validaciones correctas
# Solo agregar un comentario explicativo:

# NOTA TÉCNICA: Los endpoints PUT y DELETE mantienen validaciones en backend
# como SEGUNDA CAPA DE SEGURIDAD. Aunque el frontend deshabilita botones,
# alguien podría llamar directamente a la API, por eso validamos aquí también.
```

```
# Los endpoints POST, GET por ID, y GET activas NO necesitan cambios


# En tu archivo de rutas (main.py o routes/descripciones.py)
@app.get("/descripciones/nombres")
def obtener_nombres_descripciones(
    db: Session = Depends(get_db)
):
    try:
        # Solo ID y nombre, activas o inactivas (para filtro completo)
        nombres = db.query(models.Descripcion.id, models.Descripcion.nombre)\
                    .order_by(models.Descripcion.nombre)\
                    .limit(500)\
                    .all()

        return [
            {"value": str(desc.id), "label": desc.nombre}
            for desc in nombres
        ]
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error al obtener nombres:
{str(e)}")

# 3. OBTENER UNA DESCRIPCIÓN POR ID
@app.get("/descripciones/{descripcion_id}")
def obtener_descripcion(descripcion_id: int, db: Session = Depends(get_db)):
    """
    Obtiene una descripción específica por su ID.

    Args:
        descripcion_id: ID de la descripción
        db: Sesión de base de datos

    Returns:
        Dict con los datos de la descripción
    """
    try:
        descripcion = db.query(models.Descripcion).filter(models.Descripcion.id ==
descripcion_id).first()

        if not descripcion:
            raise HTTPException(status_code=404, detail="Descripción no encontrada")

        return descripcion.to_dict()

    except HTTPException:
        raise
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error al obtener descripción:
{str(e)}")
```

```python
# 4. ACTUALIZAR UNA DESCRIPCIÓN
@app.put("/descripciones/{descripcion_id}")
def actualizar_descripcion(
    descripcion_id: int,
    descripcion: DescripcionUpdate,
    db: Session = Depends(get_db)
):
    """
    Actualiza una descripción existente.

    Args:
        descripcion_id: ID de la descripción a actualizar
        descripcion: Datos a actualizar
        db: Sesión de base de datos

    Returns:
        Dict con la descripción actualizada
    """
    try:
        # Buscar la descripción
        db_descripcion = db.query(models.Descripcion).filter(models.Descripcion.id ==
descripcion_id).first()

        if not db_descripcion:
            raise HTTPException(status_code=404, detail="Descripción no encontrada")

        # Actualizar solo los campos que no son None
        update_data = descripcion.dict(exclude_unset=True)

        for campo, valor in update_data.items():
            setattr(db_descripcion, campo, valor)

        db.commit()
        db.refresh(db_descripcion)

        return db_descripcion.to_dict()

    except HTTPException:
        raise
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al actualizar descripción:
{str(e)}")


# 5. ELIMINAR UNA DESCRIPCIÓN
@app.delete("/descripciones/{descripcion_id}")
def eliminar_descripcion(descripcion_id: int, db: Session = Depends(get_db)):
    """
```

```
    Elimina una descripción de la base de datos.

    IMPORTANTE: Verifica que no haya movimientos o deudas asociadas antes de eliminar.

    Args:
        descripcion_id: ID de la descripción a eliminar
        db: Sesión de base de datos

    Returns:
        Dict con mensaje de confirmación
    """
    try:
        # Buscar la descripción
        descripcion = db.query(models.Descripcion).filter(models.Descripcion.id == descripcion_id).first()

        if not descripcion:
            raise HTTPException(status_code=404, detail="Descripción no encontrada")

        # Verificar si tiene movimientos asociados
        movimientos_count = db.query(models.Movimiento).filter(models.Movimiento.descripcion_id == descripcion_id).count()
        if movimientos_count > 0:
            raise HTTPException(
                status_code=400,
                detail=f"No se puede eliminar. Hay {movimientos_count} movimiento(s) asociado(s) a esta descripción."
            )

        # Verificar si tiene deudas asociadas
        deudas_count = db.query(models.Deuda).filter(models.Deuda.descripcion_id == descripcion_id).count()
        if deudas_count > 0:
            raise HTTPException(
                status_code=400,
                detail=f"No se puede eliminar. Hay {deudas_count} deuda(s) asociada(s) a esta descripción."
            )

        # Si no hay asociaciones, eliminar
        db.delete(descripcion)
        db.commit()

        return {"message": "Descripción eliminada exitosamente", "id": descripcion_id}

    except HTTPException:
        raise
    except Exception as e:
        db.rollback()
```

```
    raise HTTPException(status_code=500, detail=f"Error al eliminar descripción: {str(e)}")


# 6. OBTENER TODAS LAS DESCRIPCIONES ACTIVAS (sin paginación)
@app.get("/descripciones/activas/lista")
def obtener_descripciones_activas(
    tipo: Optional[str] = None,
    db: Session = Depends(get_db)
):
    """
    Obtiene todas las descripciones activas, útil para llenar dropdowns.

    Args:
        tipo: Filtro opcional por tipo ('ingreso', 'egreso', 'deuda')
        db: Sesión de base de datos

    Returns:
        Lista de descripciones activas
    """
    try:
        query = db.query(models.Descripcion).filter(models.Descripcion.activa == True)

        if tipo:
            query = query.filter(models.Descripcion.tipo == tipo)

        descripciones = query.order_by(models.Descripcion.nombre).all()

        return [d.to_dict() for d in descripciones]

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error al obtener descripciones activas: {str(e)}")


# REEMPLAZAR estos endpoints en main.py
@app.get("/informe/ingresos")
def informe_ingresos(
    desde: date = Query(None),
    hasta: date = Query(None),
    descripcion: str = Query(None),  # ← NUEVO PARÁMETRO
    db: Session = Depends(get_db)
):
    """
    Devuelve todos los ingresos registrados en la tabla movimientos.
    Se pueden filtrar por rango de fechas y descripción.
    """
    # JOIN con Descripcion para obtener tipo
    query = db.query(models.Movimiento).join(
        models.Descripcion,
        models.Movimiento.descripcion_id == models.Descripcion.id
```

```
    ).filter(models.Descripcion.tipo == "ingreso")

    if desde:
        query = query.filter(models.Movimiento.fecha >= desde)
    if hasta:
        query = query.filter(models.Movimiento.fecha <= hasta)

    # ← NUEVO: Filtro por descripción
    if descripcion:
        query = query.filter(models.Descripcion.nombre == descripcion)

    ingresos = query.order_by(models.Movimiento.fecha).all()
    return [ingreso.to_dict() for ingreso in ingresos]


@app.get("/informe/egresos")
def informe_egresos(
    desde: date = Query(None),
    hasta: date = Query(None),
    descripcion: str = Query(None),  # ← NUEVO PARÁMETRO
    db: Session = Depends(get_db)
):
    """
    Devuelve todos los egresos registrados en la tabla movimientos.
    Se pueden filtrar por rango de fechas y descripción.

    NUEVO: Incluye fecha de registro de deuda si el egreso está vinculado a una deuda
    """
    # JOIN con Descripcion para tipo y LEFT JOIN con Deuda para fecha_registro
    query = db.query(models.Movimiento).join(
        models.Descripcion,
        models.Movimiento.descripcion_id == models.Descripcion.id
    ).outerjoin(
        models.Deuda,
        models.Movimiento.deuda_id == models.Deuda.id
    ).filter(models.Descripcion.tipo == "egreso")

    if desde:
        query = query.filter(models.Movimiento.fecha >= desde)
    if hasta:
        query = query.filter(models.Movimiento.fecha <= hasta)

    # ← NUEVO: Filtro por descripción
    if descripcion:
        query = query.filter(models.Descripcion.nombre == descripcion)

    egresos = query.order_by(models.Movimiento.fecha).all()

    # Enriquecer con fecha de deuda
    resultado = []
    for egreso in egresos:
```

```
        egreso_dict = egreso.to_dict()
        # Agregar fecha de registro de deuda si existe
        if egreso.deuda_id and egreso.deuda:
            egreso_dict['deuda_fecha_registro'] = egreso.deuda.fecha_registro.isoformat()
        else:
            egreso_dict['deuda_fecha_registro'] = None
        resultado.append(egreso_dict)

    return resultado


@app.get("/informe/ing_egr")
def informe_ing_egr(
    desde: date = Query(None),
    hasta: date = Query(None),
    descripcion: str = Query(None),  # ← NUEVO PARÁMETRO
    db: Session = Depends(get_db)
):
    """
    Devuelve todos los movimientos (ingresos y egresos).
    Se pueden filtrar por rango de fechas y descripción.

    NUEVO: Incluye fecha de registro de deuda si el movimiento está vinculado a una deuda
    """
    from sqlalchemy.orm import joinedload

    # LEFT JOIN con Deuda usando joinedload para cargar la relación
    query = db.query(models.Movimiento).join(
        models.Descripcion,
        models.Movimiento.descripcion_id == models.Descripcion.id
    ).options(
        joinedload(models.Movimiento.deuda)
    )

    if desde:
        query = query.filter(models.Movimiento.fecha >= desde)
    if hasta:
        query = query.filter(models.Movimiento.fecha <= hasta)

    # ← NUEVO: Filtro por descripción
    if descripcion:
        query = query.filter(models.Descripcion.nombre == descripcion)

    movimientos = query.order_by(models.Movimiento.fecha).all()

    # Enriquecer con fecha de deuda
    resultado = []
    for mov in movimientos:
        mov_dict = mov.to_dict()
        # Agregar fecha de registro de deuda si existe
```

```
    if mov.deuda_id and mov.deuda:
        mov_dict['deuda_fecha_registro'] = mov.deuda.fecha_registro.isoformat()
    else:
        mov_dict['deuda_fecha_registro'] = None
    resultado.append(mov_dict)

  return resultado

# NO MODIFICAR - Este endpoint está bien como está
@app.get("/graficos/ing_egr")
def grafico_ingresos_egresos(
  desde: date = Query(..., description="Fecha de inicio"),
  hasta: date = Query(..., description="Fecha de fin"),
  db: Session = Depends(get_db)
):
  """
  Devuelve los ingresos y egresos agrupados por mes dentro del intervalo dado.
  Formato de salida ideal para gráficos.
  """
  # Ingresos agrupados por mes
  ingresos = db.query(
    extract('year', models.Movimiento.fecha).label('anio'),
    extract('month', models.Movimiento.fecha).label('mes'),
    func.sum(models.Movimiento.monto).label('total')
  ).join(
    models.Descripcion,
    models.Movimiento.descripcion_id == models.Descripcion.id
  ).filter(
    models.Descripcion.tipo == "ingreso",
    models.Movimiento.fecha >= desde,
    models.Movimiento.fecha <= hasta
  ).group_by('anio', 'mes').order_by('anio', 'mes').all()

  # Egresos agrupados por mes
  egresos = db.query(
    extract('year', models.Movimiento.fecha).label('anio'),
    extract('month', models.Movimiento.fecha).label('mes'),
    func.sum(models.Movimiento.monto).label('total')
  ).join(
    models.Descripcion,
    models.Movimiento.descripcion_id == models.Descripcion.id
  ).filter(
    models.Descripcion.tipo == "egreso",
    models.Movimiento.fecha >= desde,
    models.Movimiento.fecha <= hasta
  ).group_by('anio', 'mes').order_by('anio', 'mes').all()

  # Transformar a diccionarios para fácil unión
  ingresos_dict = { (int(a), int(m)): float(t) for a, m, t in ingresos }
  egresos_dict = { (int(a), int(m)): float(t) for a, m, t in egresos }
```

```
    # Unir todos los meses presentes en ingresos o egresos
    todos_meses = sorted(set(ingresos_dict.keys()) | set(egresos_dict.keys()))

    resultado = []
    for anio, mes in todos_meses:
        resultado.append({
            "anio": anio,
            "mes": mes,
            "ingresos": ingresos_dict.get((anio, mes), 0),
            "egresos": egresos_dict.get((anio, mes), 0)
        })

    return resultado

@app.get("/resumen-general")
def resumen_general(db: Session = Depends(get_db)):
    ahora = datetime.now()
    anio = ahora.year
    mes = ahora.month

    # Ingresos/Egresos anuales (totales) - usando JOIN con descripciones
    ingresos_total = db.query(func.sum(models.Movimiento.monto)).join(
        models.Descripcion, models.Movimiento.descripcion_id == models.Descripcion.id
    ).filter(
        models.Descripcion.tipo == "ingreso",
        extract('year', models.Movimiento.fecha) == anio
    ).scalar() or 0

    egresos_total = db.query(func.sum(models.Movimiento.monto)).join(
        models.Descripcion, models.Movimiento.descripcion_id == models.Descripcion.id
    ).filter(
        models.Descripcion.tipo == "egreso",
        extract('year', models.Movimiento.fecha) == anio
    ).scalar() or 0

    # Ingresos/Egresos por mes (barras) - usando JOIN
    ingresos_por_mes = db.query(
        extract('month', models.Movimiento.fecha).label('mes'),
        func.sum(models.Movimiento.monto)
    ).join(
        models.Descripcion, models.Movimiento.descripcion_id == models.Descripcion.id
    ).filter(
        models.Descripcion.tipo == "ingreso",
        extract('year', models.Movimiento.fecha) == anio
    ).group_by('mes').order_by('mes').all()

    egresos_por_mes = db.query(
        extract('month', models.Movimiento.fecha).label('mes'),
        func.sum(models.Movimiento.monto)
    ).join(
        models.Descripcion, models.Movimiento.descripcion_id == models.Descripcion.id
```

```
    ).filter(
        models.Descripcion.tipo == "egreso",
        extract('year', models.Movimiento.fecha) == anio
    ).group_by('mes').order_by('mes').all()

    # Deudas anuales - torta (pagadas vs pendientes) - SIN CAMBIOS
    deudas_pendientes = db.query(func.sum(models.Deuda.saldo_pendiente)).filter(
        models.Deuda.pagado == 0,
        extract('year', models.Deuda.fecha_registro) == anio
    ).scalar() or 0

    deudas_pagadas = db.query(func.sum(models.Deuda.monto)).filter(
        models.Deuda.pagado == 1,
        extract('year', models.Deuda.fecha_registro) == anio
    ).scalar() or 0

    # Deudas mensuales por tipo (barras) - SIN CAMBIOS
    deudas_mensuales_pendientes = db.query(
        extract('month', models.Deuda.fecha_registro).label("mes"),
        func.sum(models.Deuda.saldo_pendiente)
    ).filter(
        extract('year', models.Deuda.fecha_registro) == anio,
        models.Deuda.pagado == 0
    ).group_by("mes").order_by("mes").all()

    deudas_mensuales_pagadas = db.query(
        extract('month', models.Deuda.fecha_registro).label("mes"),
        func.sum(models.Deuda.monto)
    ).filter(
        extract('year', models.Deuda.fecha_registro) == anio,
        models.Deuda.pagado == 1
    ).group_by("mes").order_by("mes").all()

    return {
        "torta_ingresos_egresos_anual": {
            "ingresos": float(ingresos_total),
            "egresos": float(egresos_total)
        },
        "barras_ingresos_egresos_mensual": {
            "ingresos": [{"mes": int(m), "total": float(t)} for m, t in
ingresos_por_mes],
            "egresos": [{"mes": int(m), "total": float(t)} for m, t in egresos_por_mes]
        },
        "torta_deudas_anual": {
            "pagadas": float(deudas_pagadas),
            "pendientes": float(deudas_pendientes)
        },
        "barras_deudas_mensual": {
            "pagadas": [{"mes": int(m), "total": float(t)} for m, t in
deudas_mensuales_pagadas],
```

```
        "pendientes": [{"mes": int(m), "total": float(t)} for m, t in
deudas_mensuales_pendientes]
    }
  }
```