Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\main.py

```
# main.py
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from core.configs import settings
from utils.scheduler import tarefa_diaria, scheduler, tarefa_limpar_relatorios
import uvicorn
from contextlib import asynccontextmanager
from api.v1.endpoints.categoria import router as categoria_router
from api.v1.endpoints.setor import router as setor_router
from api.v1.endpoints.usuario import router as usuario_router
from api.v1.endpoints.item import router as item_router
from api.v1.endpoints.retirada import router as retirada_router
from api.v1.endpoints.relatorios import router as relatorio_router
from api.v1.endpoints.alerta import router as alerta_router
from fastapi.staticfiles import StaticFiles
from frontend.routes.home import router as frontend_router
import mimetypes
from utils.logger import logger

# roteador WebSocket e o manager
from utils.websocket_endpoints import websocket_router

logger.info("Iniciando aplicação Almoxarifado…")


# Forçar o tipo MIME para arquivos .js (ANTES de StaticFiles ser montado)
mimetypes.add_type('application/javascript','.js')
mimetypes.add_type('application/javascript', 'mjs') # Para módulos ES6 com extensão .mjs

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Executado antes do app iniciar
    try:
        scheduler.add_job(tarefa_diaria, 'cron', hour=9, minute=52) # verificar validade dos produtos
        scheduler.add_job(tarefa_limpar_relatorios, 'cron', hour=11,  minute=8) # limpar relatórios anti
        scheduler.start()
        print("Scheduler iniciado com sucesso via lifespan.")
    except Exception as e:
        print("Erro ao iniciar scheduler via lifespan:", e)
    yield # app roda
    # Executado quando o app estiver encerrando
    scheduler.shutdown()
    print("Scheduler finalizado.")

app = FastAPI(
    title="Sistema de Gerenciamento de Almoxarifado",
    description="API para gerenciar o estoque e retirada de materiais",
    version="1.0.0",
    lifespan=lifespan
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:8082"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
# Incluindo os endpoints Back-end
```

```
app.include_router(setor_router, prefix=settings.API_STR, tags=['Gerenciamento de Setores'])
app.include_router(categoria_router, prefix=settings.API_STR, tags=['Gerenciamento de Categorias'])
app.include_router(usuario_router, prefix=settings.API_STR, tags=['Gerenciamento de Usuários'])
app.include_router(item_router, prefix=settings.API_STR, tags=['Gerenciamento de Itens'])
app.include_router(retirada_router, prefix=settings.API_STR, tags=['Gerenciamento de Retiradas'])
app.include_router(relatorio_router, prefix=settings.API_STR, tags=['Geração de Relatórios de Itens'])
app.include_router(alerta_router, prefix=settings.API_STR, tags=['Gerenciamento de Alertas'])

# Incluir o roteador WebSocket
# Adicionado um parâmetro para o user_id no WebSocket, que será opcional na rota
app.include_router(websocket_router, prefix=settings.API_STR) # Prefixo para o WebSocket

# Montar pasta de arquivos estáticos
app.mount("/static", StaticFiles(directory="frontend/static"), name="static")

# Incluindo os endpoints Front-End
app.include_router(frontend_router)

if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8082, reload=True, log_level="debug")
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\api__init__.py

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\api\v1__init__.py

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\api\v1\endpoints\alerta.py

```
#api\vl\endpoints\alerta.py

from fastapi import APIRouter, Depends, Query, status # Importar 'status' para HTTP status codes
from sqlalchemy.ext.asyncio import AsyncSession

from core.database import get_session

from services.alerta_service import AlertaService

from core.security import usuario_almoxarifado, direcao_ou_almoxarifado, todos_usuarios
from models.alerta import TipoAlerta

from schemas.alerta import PaginatedAlertas, AlertaOut

router = APIRouter (prefix="/alertas")

@router.get("/", response_model=list [AlertaOut] , dependencies=[Depends (direcao_ou_almoxarifado)])
async def listar_todos_alertas (db: AsyncSession = Depends (get_session)):
    """Lista todos os alertas do sistema (sem paginação)."""
    return await AlertaService.get_alertas(db)

@router.get("/paginated", response_model=PaginatedAlertas, dependencies=[Depends (direcao_ou_almoxarifad
async def listar_alertas_paginados (
```

```
    page: int = Query(1, ge=1, description="Número da página"),
    size: int = Query(10, ge=1, le=100, description="Alertas por página: 5, 10, 25, 50 ou 100"),
    tipo_alerta: int | None = Query (None, description="Filtrar por tipo de alerta (1: Estoque Baixo, 2:
    search_term: str | None = Query (None, description="Filtrar por mensagem do alerta ou ID do item (pa
    db: AsyncSession = Depends (get_session)
):
    """Lista alertas do sistema com paginação e filtros."""
    return await AlertaService.get_alertas_paginated(db, page, size, tipo_alerta, search_term)

@router.patch("/ignorar/{alerta_id}", response_model=AlertaOut, dependencies=[Depends (usuario_almoxarif
async def ignorar_alerta (alerta_id: int, db: AsyncSession = Depends (get_session)):
    """Marca um alerta como 'ignorar novos', para que não seja gerado novamente para o mesmo item/motivo
    return await AlertaService.mark_alerta_as_ignorar_novos (db, alerta_id)

#  Endpoint para obter a contagem de alertas não visualizados
@router.get("/unviewed-count", dependencies=[Depends (todos_usuarios)])
async def get_unviewed_alerts_count(db: AsyncSession = Depends (get_session)):
    """Retorna o número de alertas não visualizados."""
    count = await AlertaService.get_unviewed_alerts_count(db)
    return {"count": count}

#  Endpoint para marcar todos os alertas como visualizados
@router.patch("/mark-viewed", dependencies=[Depends (direcao_ou_almoxarifado)])
async def mark_all_alerts_as_viewed (db: AsyncSession = Depends (get_session)):
    """Marca todos os alertas como visualizados."""
    await AlertaService.mark_all_alerts_as_viewed (db)
    return {"message": "Todos os alertas marcados como visualizados."}

# Deletar Alerta
@router.delete("/{alerta_id}", status_code=status.HTTP_200_OK, dependencies=[Depends(usuario_almoxarifad
async def delete_alerta(alerta_id: int, db: AsyncSession = Depends(get_session)):
    """Deleta um alerta específico pelo ID."""
    return await AlertaService.delete_alerta(db, alerta_id)
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\api\v1\endpoints\categoria.py

```
from fastapi import APIRouter, Depends, Query, status, HTTPException
from sqlalchemy.ext.asyncio import AsyncSession
from schemas.categoria import CategoriaCreate, CategoriaUpdate, CategoriaOut, PaginatedCategorias
from services.categoria_service import CategoriaService
from typing import List
from core.database import get_session
from core.security import usuario_almoxarifado, direcao_ou_almoxarifado
from utils.logger import logger

router = APIRouter(prefix="/categorias")


@router.post("/", response_model=CategoriaOut, status_code=status.HTTP_201_CREATED)
async def create_categoria(
    categoria: CategoriaCreate,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(usuario_almoxarifado)
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} criando categoria: {categoria.nome_categoria}")
        return await CategoriaService.create_categoria(db, categoria)
    except Exception as e:
        logger.error(f"Erro ao criar categoria: {categoria.nome_categoria} | {e}")
        raise HTTPException(status_code=500, detail="Erro ao criar categoria")
```

```
@router.get("/buscar", response_model=PaginatedCategorias, dependencies=[Depends(usuario_almoxarifado)])
async def search_categorias(
    nome: str | None = Query(None),
    page: int = Query(1, ge=1),
    size: int = Query(10),
    db: AsyncSession = Depends(get_session),
    current_user=Depends(direcao_ou_almoxarifado)
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} buscando categorias (nome={nome}, page={page}, s
        return await CategoriaService.search_categorias_paginated(db, nome_categoria=nome, page=page, si
    except Exception as e:
        logger.error(f"Erro ao buscar categorias: {e}")
        raise HTTPException(status_code=500, detail="Erro ao buscar categorias")


@router.get("/paginated", response_model=PaginatedCategorias, dependencies=[Depends(direcao_ou_almoxarif
async def get_items_paginated(
    page: int = Query(1, ge=1),
    size: int = Query(10),
    db: AsyncSession = Depends(get_session)
):
    try:
        logger.info(f"Listando categorias paginadas (page={page}, size={size})")
        return await CategoriaService.get_categorias_paginated(db, page, size)
    except Exception as e:
        logger.error(f"Erro ao listar categorias paginadas: {e}")
        raise HTTPException(status_code=500, detail="Erro ao listar categorias")


@router.get("/", response_model=List[CategoriaOut])
async def get_categorias(db: AsyncSession = Depends(get_session), current_user=Depends(direcao_ou_almoxa
    try:
        logger.info(f"Usuário {current_user.usuario_id} listando todas as categorias")
        return await CategoriaService.get_categorias(db)
    except Exception as e:
        logger.error(f"Erro ao listar categorias: {e}")
        raise HTTPException(status_code=500, detail="Erro ao listar categorias")


@router.get("/{categoria_id}", response_model=CategoriaOut)
async def get_categoria_by_id(categoria_id: int, db: AsyncSession = Depends(get_session), current_user=D
    try:
        logger.info(f"Usuário {current_user.usuario_id} consultando categoria por ID: {categoria_id}")
        return await CategoriaService.get_categoria_by_id(db, categoria_id)
    except Exception as e:
        logger.error(f"Erro ao buscar categoria ID {categoria_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao buscar categoria")


@router.get("/{categoria_name}", response_model=CategoriaOut)
async def get_categoria_by_name(categoria_name: str, db: AsyncSession = Depends(get_session), current_us
    try:
        logger.info(f"Usuário {current_user.usuario_id} consultando categoria por nome: {categoria_name}
        return await CategoriaService.get_categoria_by_name(db, categoria_name)
    except Exception as e:
        logger.error(f"Erro ao buscar categoria '{categoria_name}': {e}")
        raise HTTPException(status_code=500, detail="Erro ao buscar categoria")


@router.put("/{categoria_id}", response_model=CategoriaUpdate)
async def update_categoria(
    categoria_id: int,
```

```
    categoria: CategoriaUpdate,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(usuario_almoxarifado)
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} atualizando categoria ID {categoria_id} para: {c
        return await CategoriaService.update_categoria(db, categoria_id, categoria)
    except Exception as e:
        logger.error(f"Erro ao atualizar categoria ID {categoria_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao atualizar categoria")


@router.delete("/{categoria_id}")
async def delete_categoria(categoria_id: int, db: AsyncSession = Depends(get_session), current_user=Depe
    try:
        logger.info(f"Usuário {current_user.usuario_id} deletando categoria ID {categoria_id}")
        return await CategoriaService.delete_categoria(db, categoria_id)
    except Exception as e:
        logger.error(f"Erro ao deletar categoria ID {categoria_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao deletar categoria")
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\api\v1\endpoints\item.py

```
from fastapi import APIRouter, Depends, status, Query, UploadFile, File, HTTPException
from sqlalchemy.ext.asyncio import AsyncSession
from typing import List
from core.database import get_session
from core.security import usuario_almoxarifado, direcao_ou_almoxarifado, todos_usuarios
from schemas.item import (
    ItemOut,
    ItemCreate,
    ItemUpdate,
    PaginatedItems,
    BulkItemUploadResult,
)
from services.item_service import ItemService
from utils.logger import logger

router = APIRouter(prefix="/itens")


@router.post("/", response_model=ItemOut, status_code=status.HTTP_201_CREATED)
async def create_item(item: ItemCreate, db: AsyncSession = Depends(get_session), current_user=Depends(us
    try:
        logger.info(f"Usuário {current_user.usuario_id} criando item: {item.nome_item}")
        return await ItemService.create_item(db, item, current_user)
    except Exception as e:
        logger.error(f"Erro ao criar item '{item.nome_item}': {e}")
        raise HTTPException(status_code=500, detail="Erro ao criar item")


@router.get("/buscar", response_model=PaginatedItems)
async def search_items(
    nome: str | None = Query(None),
    categoria: str | None = Query(None),
    page: int = Query(1, ge=1),
    size: int = Query(10),
    db: AsyncSession = Depends(get_session),
    current_user=Depends(todos_usuarios),
):
```

```
    try:
        logger.info(f"Usuário {current_user.usuario_id} buscando itens (nome={nome}, categoria={categori
        return await ItemService.search_items_paginated(db, nome_produto=nome, nome_categoria=categoria,
    except Exception as e:
        logger.error(f"Erro ao buscar itens: {e}")
        raise HTTPException(status_code=500, detail="Erro ao buscar itens")


@router.get("/paginated", response_model=PaginatedItems, dependencies=[Depends(todos_usuarios)])
async def get_items_paginated(
    page: int = Query(1, ge=1),
    size: int = Query(10),
    db: AsyncSession = Depends(get_session),
):
    try:
        logger.info(f"Listando itens paginados (page={page}, size={size})")
        return await ItemService.get_items_paginated(db, page, size)
    except Exception as e:
        logger.error(f"Erro ao listar itens paginados: {e}")
        raise HTTPException(status_code=500, detail="Erro ao listar itens")


@router.get("/", response_model=List[ItemOut])
async def get_itens(db: AsyncSession = Depends(get_session), current_user=Depends(direcao_ou_almoxarifad
    try:
        logger.info(f"Usuário {current_user.usuario_id} listando todos os itens")
        return await ItemService.get_itens(db)
    except Exception as e:
        logger.error(f"Erro ao listar itens: {e}")
        raise HTTPException(status_code=500, detail="Erro ao listar itens")


@router.get("/{item_id}", response_model=ItemOut)
async def get_item(item_id: int, db: AsyncSession = Depends(get_session), current_user=Depends(todos_usu
    try:
        logger.info(f"Usuário {current_user.usuario_id} consultando item ID {item_id}")
        return await ItemService.get_item_by_id(db, item_id)
    except Exception as e:
        logger.error(f"Erro ao buscar item ID {item_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao buscar item")


@router.delete("/{item_id}")
async def delete_item(item_id: int, db: AsyncSession = Depends(get_session), current_user=Depends(usuari
    try:
        logger.info(f"Usuário {current_user.usuario_id} deletando item ID {item_id}")
        return await ItemService.delete_item(db, item_id)
    except Exception as e:
        logger.error(f"Erro ao deletar item ID {item_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao deletar item")


@router.put("/{item_id}", response_model=ItemOut)
async def update_item(item_id: int, item: ItemUpdate, db: AsyncSession = Depends(get_session), current_u
    try:
        logger.info(f"Usuário {current_user.usuario_id} atualizando item ID {item_id} para: {item.nome_i
        return await ItemService.update_item(db, item_id, item, current_user)
    except Exception as e:
        logger.error(f"Erro ao atualizar item ID {item_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao atualizar item")


@router.post("/upload-bulk/", response_model=BulkItemUploadResult)
```

```
async def upload_items_bulk(file: UploadFile = File(...), db: AsyncSession = Depends(get_session), curre
    try:
        logger.info(f"Usuário {current_user.usuario_id} fez upload de arquivo de itens: {file.filename}"
        return await ItemService.process_bulk_upload(db, file, current_user.usuario_id)
    except Exception as e:
        logger.error(f"Erro no upload em massa de itens: {e}")
        raise HTTPException(status_code=500, detail="Erro ao processar upload de itens")
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\api\v1\endpoints\relatorios.py

```
from fastapi import APIRouter, Depends, Query, HTTPException
from sqlalchemy.ext.asyncio import AsyncSession
from fastapi.responses import FileResponse
from datetime import datetime
import os
from core.database import get_session
from services.relatorio_service import RelatorioService
from core.security import direcao_ou_almoxarifado
from utils.logger import logger

router = APIRouter()


@router.get("/relatorios/quantidade-itens/")
async def gerar_relatorio_quantidade(
    filtro_categoria: str = Query(None),
    filtro_produto: str = Query(None),
    formato: str = Query("csv"),
    db: AsyncSession = Depends(get_session),
    current_user=Depends(direcao_ou_almoxarifado)
):
    try:
        logger.info(
            f"Usuário {current_user.usuario_id} solicitou relatório quantidade-itens "
            f"(categoria={filtro_categoria}, produto={filtro_produto}, formato={formato})"
        )
        caminho = await RelatorioService.gerar_relatorio_quantidade_itens(
            db, filtro_categoria, filtro_produto, formato
        )
        if not caminho or not os.path.exists(str(caminho)):
            logger.warning("Arquivo de relatório quantidade-itens não encontrado após geração")
            raise HTTPException(status_code=404, detail="Relatório não gerado")
        logger.info(f"Relatório quantidade-itens gerado em: {caminho}")
        return FileResponse(
            path=str(caminho),
            filename=os.path.basename(str(caminho)),
            media_type="application/octet-stream"
        )
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Erro ao gerar relatório quantidade-itens: {e}")
        raise HTTPException(status_code=500, detail="Erro ao gerar relatório")


@router.get("/relatorios/entrada-itens/")
async def gerar_relatorio_entrada(
    data_inicio: datetime = Query(...),
    data_fim: datetime = Query(...),
    formato: str = Query("csv"),
```

```
    db: AsyncSession = Depends(get_session),
    current_user=Depends(direcao_ou_almoxarifado)
):
    try:
        logger.info(
            f"Usuário {current_user.usuario_id} solicitou relatório entrada-itens "
            f"(de={data_inicio.date()}, até={data_fim.date()}, formato={formato})"
        )
        caminho = await RelatorioService.gerar_relatorio_entrada_itens(
            db, data_inicio, data_fim, formato
        )
        if not caminho or not os.path.exists(str(caminho)):
            logger.warning("Arquivo de relatório entrada-itens não encontrado após geração")
            raise HTTPException(status_code=404, detail="Relatório não gerado")
        logger.info(f"Relatório entrada-itens gerado em: {caminho}")
        return FileResponse(
            path=str(caminho),
            filename=os.path.basename(str(caminho)),
            media_type="application/octet-stream"
        )
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Erro ao gerar relatório entrada-itens: {e}")
        raise HTTPException(status_code=500, detail="Erro ao gerar relatório")



@router.get("/relatorios/retiradas-setor/")
async def gerar_relatorio_retiradas_setor(
    setor_id: int = Query(...),
    data_inicio: datetime = Query(...),
    data_fim: datetime = Query(...),
    formato: str = Query("csv"),
    db: AsyncSession = Depends(get_session),
    current_user=Depends(direcao_ou_almoxarifado)
):
    try:
        logger.info(
            f"Usuário {current_user.usuario_id} solicitou relatório retiradas-setor "
            f"(setor_id={setor_id}, de={data_inicio.date()}, até={data_fim.date()}, formato={formato})"
        )
        caminho = await RelatorioService.gerar_relatorio_retiradas_setor(
            db, setor_id, data_inicio, data_fim, formato
        )
        if not caminho or not os.path.exists(str(caminho)):
            logger.warning("Arquivo de relatório retiradas-setor não encontrado após geração")
            raise HTTPException(status_code=404, detail="Relatório não gerado")
        logger.info(f"Relatório retiradas-setor gerado em: {caminho}")
        return FileResponse(path=str(caminho), filename=os.path.basename(str(caminho)), media_type="appl
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Erro ao gerar relatório retiradas-setor: {e}")
        raise HTTPException(status_code=500, detail="Erro ao gerar relatório")


@router.get("/relatorios/retiradas-usuario/")
async def gerar_relatorio_retiradas_usuario(
    usuario_id: int = Query(...),
    data_inicio: datetime = Query(...),
    data_fim: datetime = Query(...),
    formato: str = Query("csv"),
```

```
    db: AsyncSession = Depends(get_session),
    current_user=Depends(direcao_ou_almoxarifado)
):
    try:
        logger.info(
            f"Usuário {current_user.usuario_id} solicitou relatório retiradas-usuario "
            f"(usuario_id={usuario_id}, de={data_inicio.date()}, até={data_fim.date()}, formato={formato
        )
        caminho = await RelatorioService.gerar_relatorio_retiradas_usuario(
            db, usuario_id, data_inicio, data_fim, formato
        )
        if not caminho or not os.path.exists(str(caminho)):
            logger.warning("Arquivo de relatório retiradas-usuario não encontrado após geração")
            raise HTTPException(status_code=404, detail="Relatório não gerado")
        logger.info(f"Relatório retiradas-usuario gerado em: {caminho}")
        return FileResponse(path=str(caminho), filename=os.path.basename(str(caminho)), media_type="appl
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Erro ao gerar relatório retiradas-usuario: {e}")
        raise HTTPException(status_code=500, detail="Erro ao gerar relatório")
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\api\v1\endpoints\retirada.py

```
#api/v1/endpoints/retirada.py

from datetime import datetime
from fastapi import APIRouter, Depends, status, Query, HTTPException
from sqlalchemy.ext.asyncio import AsyncSession

from core.database import get_session
from schemas.retirada import (
    RetiradaCreate, RetiradaUpdateStatus, RetiradaOut,
    RetiradaPaginated, RetiradaFilterParams, StatusEnum
)
from services.retirada_service import RetiradaService
from core.security import todos_usuarios, usuario_almoxarifado, direcao_ou_almoxarifado
from utils.logger import logger

router = APIRouter(prefix="/retiradas")

@router.post("/", response_model=RetiradaOut, status_code=status.HTTP_201_CREATED)
async def solicitar_retirada(
    retirada: RetiradaCreate,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(todos_usuarios)
):
    """Endpoint para um usuário solicitar uma nova retirada de itens."""
    try:
        logger.info(f"Usuário {current_user.username} solicitou uma retirada de itens")
        return await RetiradaService.solicitar_retirada(db, retirada, current_user.usuario_id)
    except Exception as e:
        logger.error(f"Erro ao solicitar retirada: {e}")
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail="Erro ao solicitar



@router.put("/{retirada_id}", response_model=RetiradaOut)
async def atualizar_status_retirada(
    retirada_id: int,
```

```
    status_data: RetiradaUpdateStatus,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(usuario_almoxarifado)
):
    """Endpoint para um usuário do almoxarifado atualizar o status de uma retirada."""
    try:
        logger.info(f"Usuário {current_user.usuario_id} atualizou status da retirada {retirada_id}")
        return await RetiradaService.atualizar_status(db, retirada_id, status_data, current_user.usuario
    except Exception as e:
        logger.error(f"Erro ao atualizar status da retirada {retirada_id}")
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail="Erro ao atualizar

# Listagem paginada de todas as retiradas (para almoxarifado)
@router.get(
    "/paginated",
    response_model=RetiradaPaginated,
    name="Listar retiradas paginadas"
)
async def listar_retiradas_paginadas(
    page: int = Query(1, ge=1),
    page_size: int = Query(10, ge=1, le=100),
    db: AsyncSession = Depends(get_session),
    current_user=Depends(direcao_ou_almoxarifado)
):
    """Lista todas as retiradas com paginação. Apenas para usuários do almoxarifado."""
    return await RetiradaService.get_retiradas_paginadas(db, page, page_size)

# Listagem paginada de retiradas pendentes (para todos os usuários, mas com filtro de permissão)
@router.get(
    "/pendentes/paginated",
    response_model=RetiradaPaginated,
    name="Listar pendentes paginados"
)
async def listar_pendentes_paginados(
    page: int = Query(1, ge=1),
    page_size: int = Query(10, ge=1, le=100),
    db: AsyncSession = Depends(get_session),
    current_user=Depends(todos_usuarios) # Pode ser acessado por todos, mas o serviço deve filtrar
):
    """Lista retiradas pendentes com paginação."""
    return await RetiradaService.get_retiradas_pendentes_paginated(db, page, page_size)

# Busca com filtros e paginação (para almoxarifado)
@router.get(
    "/search",
    response_model=RetiradaPaginated,
    name="Buscar retiradas por filtros paginados"
)
async def buscar_retiradas(
    status: StatusEnum | None = Query(None),
    solicitante: str | None = Query(None),
    start_date: datetime | None = Query(None),
    end_date: datetime | None = Query(None),
    page: int = Query(1, ge=1),
    page_size: int = Query(10, ge=1, le=100),
    db: AsyncSession = Depends(get_session),
    current_user=Depends(direcao_ou_almoxarifado)
):
    """Busca retiradas com filtros e paginação. Apenas para usuários do almoxarifado."""
    params = RetiradaFilterParams(
        status=status,
        solicitante=solicitante,
        start_date=start_date,
```

```
        end_date=end_date,
    )
    return await RetiradaService.filter_retiradas_paginated(db, params, page, page_size)

# Recupera uma específica (para almoxarifado)
@router.get("/{retirada_id}", response_model=RetiradaOut)
async def get_retirada(
    retirada_id: int,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(direcao_ou_almoxarifado)
):
    """Recupera uma retirada específica pelo ID. Apenas para usuários do almoxarifado."""
    return await RetiradaService.get_retirada_by_id(db, retirada_id)

# Listagem paginada das minhas retiradas (para servidor)
@router.get(
    "/minhas-retiradas/paginated",
    response_model=RetiradaPaginated,
    name="Listar minhas retiradas paginadas"
)
async def listar_minhas_retiradas_paginadas(
    page: int = Query(1, ge=1),
    page_size: int = Query(10, ge=1, le=100),
    db: AsyncSession = Depends(get_session),
    current_user=Depends(todos_usuarios) # Acessível por qualquer usuário logado
):
    """Lista as retiradas solicitadas pelo usuário logado, com paginação."""
    return await RetiradaService.get_retiradas_by_user_paginated(db, current_user.usuario_id, page, page

# Soft delete de retiradas por período
@router.delete(
    "/soft-delete-by-period",
    status_code=status.HTTP_200_OK,
    dependencies=[Depends(direcao_ou_almoxarifado)], # Apenas Direção ou Almoxarifado pode fazer isso
    summary="Deleta (inativa) retiradas antigas por período"
)
async def soft_delete_retiradas(
    start_date: datetime = Query(..., description="Data inicial do período (YYYY-MM-DD)"),
    end_date: datetime = Query(..., description="Data final do período (YYYY-MM-DD)"),
    db: AsyncSession = Depends(get_session)
):
    """
    Deleta (inativa) logicamente retiradas antigas em um período especificado.
    As retiradas não são removidas fisicamente, apenas marcadas como inativas.
    """
    return await RetiradaService.soft_delete_retiradas_by_period(db, start_date, end_date)
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\api\v1\endpoints\setor.py

```
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.ext.asyncio import AsyncSession
from core.database import get_session
from schemas.setor import SetorCreate, SetorUpdate, SetorOut
from services.setor_service import SetorService
from typing import List
from core.security import usuario_direcao, todos_usuarios
from utils.logger import logger

router = APIRouter(prefix="/setores")
@router.post("/", response_model=SetorOut, status_code=status.HTTP_201_CREATED)
```

```
async def create_setor(
    setor: SetorCreate,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(usuario_direcao)
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} criando setor: {setor.nome_setor}")
        return await SetorService.create_setor(db, setor)
    except Exception as e:
        logger.error(f"Erro ao criar setor '{setor.nome_setor}': {e}")
        raise HTTPException(status_code=500, detail="Erro ao criar setor")


@router.get("/", response_model=List[SetorOut], status_code=status.HTTP_200_OK)
async def get_setores(
    db: AsyncSession = Depends(get_session),
    current_user=Depends(todos_usuarios)
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} listando todos os setores")
        return await SetorService.get_setores(db)
    except Exception as e:
        logger.error(f"Erro ao listar setores: {e}")
        raise HTTPException(status_code=500, detail="Erro ao listar setores")


@router.get("/{setor_id}", response_model=SetorOut, status_code=status.HTTP_200_OK)
async def get_setor_by_id(
    setor_id: int,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(todos_usuarios)
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} consultando setor ID {setor_id}")
        setor = await SetorService.get_setor_by_id(db, setor_id)
        if not setor:
            logger.warning(f"Setor ID {setor_id} não encontrado")
            raise HTTPException(status_code=404, detail="Setor não encontrado")
        return setor
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Erro ao buscar setor ID {setor_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao buscar setor")


@router.put("/{setor_id}", response_model=SetorOut, status_code=status.HTTP_200_OK)
async def update_setor(
    setor_id: int,
    setor: SetorUpdate,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(usuario_direcao)
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} atualizando setor ID {setor_id} para: {setor.nom
        updated = await SetorService.update_setor(db, setor_id, setor)
        if not updated:
            logger.warning(f"Tentativa de atualizar setor ID {setor_id} não existente")
            raise HTTPException(status_code=404, detail="Setor não encontrado")
        return updated
    except HTTPException:
        raise
    except Exception as e:
```

```
        logger.error(f"Erro ao atualizar setor ID {setor_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao atualizar setor")


@router.delete("/{setor_id}", status_code=status.HTTP_200_OK)
async def delete_setor(
    setor_id: int,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(usuario_direcao)
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} deletando setor ID {setor_id}")
        deleted = await SetorService.delete_setor(db, setor_id)
        if not deleted:
            logger.warning(f"Tentativa de deletar setor ID {setor_id} não existente")
            raise HTTPException(status_code=404, detail="Setor não encontrado")
        return {"message": "Setor deletado com sucesso"}
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Erro ao deletar setor ID {setor_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao deletar setor")
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\api\v1\endpoints\usuario.py

```
# api/v1/endpoints/usuario.py

from fastapi import APIRouter, Depends, status, Response, HTTPException
from fastapi.security import OAuth2PasswordRequestForm
from sqlalchemy.ext.asyncio import AsyncSession
from typing import List
from core.database import get_session
from core.security import usuario_direcao, direcao_ou_almoxarifado, todos_usuarios
from schemas.usuario import (
    UsuarioOut,
    UsuarioCreate,
    UsuarioUpdate,
    UsuarioResetPasswordSimple,
    UsuarioCheckForReset,
)
from schemas.auth_schemas import TokenSchema
from services.usuario_service import UsuarioService
from utils.logger import logger

router = APIRouter(prefix="/usuarios")


@router.post("/primeiro-usuario", status_code=status.HTTP_201_CREATED, response_model=UsuarioOut)
async def criar_primeiro_usuario(
    usuario_data: UsuarioCreate,
    db: AsyncSession = Depends(get_session),
):
    try:
        logger.info("Criando primeiro usuário")
        return await UsuarioService.create_first_user(db, usuario_data)
    except Exception as e:
        logger.error(f"Erro ao criar primeiro usuário: {e}")
        raise HTTPException(status_code=500, detail="Erro ao criar primeiro usuário")
@router.post("/", response_model=UsuarioOut, status_code=status.HTTP_201_CREATED)
async def create_user(
```

```
    user: UsuarioCreate,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(usuario_direcao),
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} criando novo usuário: {user.username}")
        return await UsuarioService.create_usuario(db, user)
    except HTTPException as http_exc: # Captura HTTPExceptions (como as de validação)
        logger.warning(f"Falha na validação ao criar usuário '{user.username}': {http_exc.detail}")
        raise http_exc # Re-lança a exceção original
    except Exception as e:
        logger.error(f"Erro inesperado ao criar usuário '{user.username}': {e}")
        raise HTTPException(status_code=500, detail="Erro ao criar usuário")


@router.get("/", response_model=List[UsuarioOut], status_code=status.HTTP_200_OK)
async def get_usuarios(
    db: AsyncSession = Depends(get_session),
    current_user=Depends(direcao_ou_almoxarifado),
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} listando todos os usuários")
        return await UsuarioService.get_usuarios(db)
    except Exception as e:
        logger.error(f"Erro ao listar usuários: {e}")
        raise HTTPException(status_code=500, detail="Erro ao listar usuários")


@router.get("/{usuario_id}", response_model=UsuarioOut, status_code=status.HTTP_200_OK)
async def get_usuario(
    usuario_id: int,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(todos_usuarios),
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} consultando usuário ID {usuario_id}")
        user = await UsuarioService.get_usuario_by_id(db, usuario_id)
        if not user:
            logger.warning(f"Usuário ID {usuario_id} não encontrado")
            raise HTTPException(status_code=404, detail="Usuário não encontrado")
        return user
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Erro ao buscar usuário ID {usuario_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao buscar usuário")


@router.delete("/{usuario_id}", status_code=status.HTTP_200_OK)
async def delete_usuario(
    usuario_id: int,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(usuario_direcao),
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} deletando usuário ID {usuario_id}")
        deleted = await UsuarioService.delete_usuario(db, usuario_id, current_user)
        if not deleted:
            logger.warning(f"Tentativa de deletar usuário ID {usuario_id} não existente")
            raise HTTPException(status_code=404, detail="Usuário não encontrado")
        return {"message": "Usuário deletado com sucesso"}
    except HTTPException:
        raise
```

```
    except Exception as e:
        logger.error(f"Erro ao deletar usuário ID {usuario_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao deletar usuário")


@router.put("/{usuario_id}", response_model=UsuarioOut, status_code=status.HTTP_200_OK)
async def update_usuario(
    usuario_id: int,
    usuario: UsuarioUpdate,
    db: AsyncSession = Depends(get_session),
    current_user=Depends(todos_usuarios),
):
    try:
        logger.info(f"Usuário {current_user.usuario_id} atualizando usuário ID {usuario_id}")
        updated = await UsuarioService.update_usuario(db, usuario_id, usuario, current_user)
        if not updated:
            logger.warning(f"Tentativa de atualizar usuário ID {usuario_id} não existente")
            raise HTTPException(status_code=404, detail="Usuário não encontrado")
        return updated
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Erro ao atualizar usuário ID {usuario_id}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao atualizar usuário")


@router.post("/token", response_model=TokenSchema)
async def get_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
    db: AsyncSession = Depends(get_session),
):
    try:
        logger.info(f"Tentativa de login para username={form_data.username}")
        token = await UsuarioService.login_user(form_data, db)
        logger.info(f"Login bem■sucedido para username={form_data.username}")
        return token
    except HTTPException as e:
        logger.warning(f"Falha no login para username={form_data.username}: {e.detail}")
        raise
    except Exception as e:
        logger.error(f"Erro inesperado no login de username={form_data.username}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao efetuar login")


@router.post("/logout", status_code=status.HTTP_204_NO_CONTENT)
def logout(response: Response):
    """
    Realiza o logout do usuário, removendo o token de acesso do cookie.
    """
    logger.info("Logout realizado com sucesso")

    # IMPORTANTE: Deleta o cookie 'access_token' explicitamente
    # Certifique-se de que 'path' e 'samesite' correspondam aos valores
    # usados quando o cookie foi definido em login.js.
    response.delete_cookie(
        "access_token",
        path="/",        # Deve corresponder ao path usado ao definir o cookie
        samesite="lax"   # Deve corresponder ao samesite usado ao definir o cookie
        # domain=None,   # Se você definiu um domínio específico ao criar o cookie,
                         # deve especificá-lo aqui também. Caso contrário, deixe None
                         # para o domínio atual (padrão).
    )
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

```
@router.post("/reset-password-simple", status_code=status.HTTP_200_OK)
async def reset_password_simple(
    data: UsuarioResetPasswordSimple,
    db: AsyncSession = Depends(get_session),
):
    try:
        logger.info(f"Reset de senha solicitado para: {data.username_or_email}")
        await UsuarioService.reset_password_simple(db, data.username_or_email, data.new_password)
        logger.info(f"Senha redefinida com sucesso para: {data.username_or_email}")
        return {"message": "Senha redefinida com sucesso!"}
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Erro ao redefinir senha para {data.username_or_email}: {e}")
        raise HTTPException(status_code=500, detail="Erro ao redefinir senha")


@router.post("/check-user-for-reset", status_code=status.HTTP_200_OK)
async def check_user_for_reset(
    data: UsuarioCheckForReset,
    db: AsyncSession = Depends(get_session),
):
    try:
        exists = await UsuarioService.check_user_exists_for_reset(db, data.username_or_email)
        if not exists:
            logger.warning(f"Usuário não encontrado para reset: {data.username_or_email}")
            raise HTTPException(status_code=404, detail="Usuário não encontrado")
        logger.info(f"Usuário confirmado para reset: {data.username_or_email}")
        return {"message": "Usuário encontrado. Prossiga para a redefinição de senha."}
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Erro ao verificar usuário para reset: {e}")
        raise HTTPException(status_code=500, detail="Erro ao verificar usuário")
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\core\configs.py

```
from sqlalchemy.ext.declarative import declarative_base
from core.config_loader import ConfigLoader
import pytz
from pathlib import Path

class Settings:
    API_STR: str = "/api/almoxarifado"

    # Banco de Dados
    DATABASE_URL: str = ConfigLoader.get("DATABASE_URL", required=True)
    DBBaseModel = declarative_base()

    # Autenticação
    JWT_SECRET: str = ConfigLoader.get("JWT_SECRET", required=True)
    ALGORITHM: str = ConfigLoader.get("ALGORITHM")
    ACCESS_TOKEN_EXPIRE_MINUTES: int = int(ConfigLoader.get("ACCESS_TOKEN_EXPIRE_MINUTES"))

    # Timezone
    BRASILIA_TIMEZONE = pytz.timezone("America/Sao_Paulo")

    # Files (caminho relativo à raiz do projeto)
    PROJECT_ROOT: Path = Path(__file__).resolve().parent.parent # Ajustar o parent conforme estrutura do
    PASTA_RELATORIOS: Path = PROJECT_ROOT / "relatorios"
```

```
    PASTA_RELATORIOS.mkdir(exist_ok=True) # Cria a pasta se não existir.

    #configuração para retenção de relatórios (em dias)
    REPORT_RETENTION_DAYS: int = int(ConfigLoader.get("REPORT_RETENTION_DAYS", default=30))

settings = Settings()
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\core\config_loader.py

```
#core\config_loader.py

import os

class ConfigLoader:
    _config = {}

    @classmethod
    def load_config(cls, file_path=".env"):
        """Carrega variáveis de um arquivo `.env` para a memória."""
        if not os.path.exists(file_path):
            print(f"[WARNING] Arquivo {file_path} não encontrado. Variáveis de ambiente devem ser config
            return

        with open(file_path, "r") as file:
            for line in file:
                line = line.strip()
                if not line or line.startswith("#"):
                    continue
                key, value = line.split("=", 1)
                cls._config[key.strip()] = value.strip()

    @classmethod
    def get(cls, key, default=None, required=False):
        """Obtém um valor do arquivo de configuração, com opção de valor padrão e obrigatoriedade."""
        value = cls._config.get(key, os.getenv(key, default))
        if required and value is None:
            raise ValueError(f"Variável de ambiente obrigatória {key} não foi encontrada!")
        return value

# Carregar configurações na inicialização
ConfigLoader.load_config()
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\core\database.py

```
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker
from core.configs import settings
from contextlib import asynccontextmanager
from sqlalchemy.ext.asyncio import async_session
from typing import AsyncGenerator


# Criar engine assíncrona
engine = create_async_engine(settings.DATABASE_URL, echo=False)

# Criar sessão assíncrona
```

```
SessionLocal = sessionmaker(
    autocommit=False,
    autoflush=False,
    expire_on_commit=False,
    class_=AsyncSession,
    bind=engine
)

# Função para obter sessão do banco de dados para usuarios
async def get_session():
    async with SessionLocal() as session:
        yield session



# Função para obter sessão do banco de dados para o scheduler
@asynccontextmanager
async def get_session_scheduler() -> AsyncGenerator[AsyncSession, None]:
    async with SessionLocal() as session:
        yield session
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\core\security.py

```
# core/security.py
from pwdlib import PasswordHash
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from jwt import encode, decode
from jwt.exceptions import PyJWTError
from datetime import datetime, timedelta
from core.configs import settings
from core.database import get_session
from sqlalchemy.ext.asyncio import AsyncSession
from models import Usuario
from sqlalchemy.future import select
from models.usuario import RoleEnum
from typing import List

pwd_context = PasswordHash.recommended()
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/api/almoxarifado/usuarios/token")

def get_password_hash(password: str):
    return pwd_context.hash(password)

def verify_password(original_password: str, hashed_password: str):
    return pwd_context.verify(original_password, hashed_password)

def create_access_token(data_payload: dict, tipo_usuario: int, usuario_id: int = None): # Adicionado usu
    to_encode = data_payload.copy()
    if tipo_usuario is not None:
        to_encode.update({'tipo_usuario': tipo_usuario})
    if usuario_id is not None: # Adiciona o usuario_id ao payload
        to_encode.update({'usuario_id': usuario_id})

    #ver se é melhor usar ZoneInfo
    expire_date = datetime.now(tz=settings.BRASILIA_TIMEZONE) + timedelta(minutes=settings.ACCESS_TOKEN_
    to_encode.update({'exp': expire_date.timestamp()})

    encoded_jwt = encode(to_encode, settings.JWT_SECRET, algorithm=settings.ALGORITHM)
    return encoded_jwt
```

```
async def get_current_user(token: str = Depends(oauth2_scheme),
                           db: AsyncSession = Depends(get_session)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail='Não foi possível validar as credenciais',
        headers={'WWW-Authenticate': 'Bearer'}
    )
    try:
        payload = decode(token, settings.JWT_SECRET, algorithms=[settings.ALGORITHM])
        username: str = payload.get('sub')
        tipo_usuario = payload.get('tipo_usuario') # Agora buscando 'tipo_usuario'
        #  Obtendo o usuario_id do token
        usuario_id = payload.get('usuario_id')

        if not username:
            raise credentials_exception
    except PyJWTError:
        raise credentials_exception

    user = await db.scalar(select(Usuario).where(Usuario.username == username))
    if not user:
        raise credentials_exception
    user.tipo_usuario_from_token = tipo_usuario
    user.usuario_id_from_token = usuario_id # Armazena o ID do token no objeto do usuário

    return user

def verify_user_type(allowed_types: List[RoleEnum]):
    allowed_values = [t.value for t in allowed_types]
    def verifier(current_user: Usuario = Depends(get_current_user)):
        tipo_usuario = current_user.tipo_usuario_from_token
        if tipo_usuario not in allowed_values:
            allowed_names = [t.name for t in allowed_types]
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail=f"Acesso restrito a: {', '.join(allowed_names)}"
            )
        return current_user
    return verifier

# Verificadores (use os mesmos nomes do RoleEnum)
usuario_direcao = verify_user_type([RoleEnum.USUARIO_DIRECAO])
usuario_almoxarifado = verify_user_type([RoleEnum.USUARIO_ALMOXARIFADO])
usuario_geral = verify_user_type([RoleEnum.USUARIO_GERAL])
direcao_ou_almoxarifado = verify_user_type([RoleEnum.USUARIO_ALMOXARIFADO, RoleEnum.USUARIO_DIRECAO])
todos_usuarios = verify_user_type([RoleEnum.USUARIO_ALMOXARIFADO, RoleEnum.USUARIO_DIRECAO, RoleEnum.USU
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\core__init__.py

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\models\alerta.py

```
#models\alerta.py

from sqlalchemy import Column, Integer, ForeignKey, TIMESTAMP, String, Boolean
from core.configs import settings
```

```
from datetime import datetime
from enum import Enum

class TipoAlerta(Enum):
    ESTOQUE_BAIXO = 1
    VALIDADE_PROXIMA = 2

class Alerta(settings.DBBaseModel):
    __tablename__ = "alerta"

    alerta_id = Column(Integer, primary_key=True, index=True)
    tipo_alerta = Column(Integer, nullable=False)
    item_id = Column(Integer, ForeignKey("item.item_id"), nullable=False)
    data_alerta = Column(TIMESTAMP, nullable=False, default=datetime.now)
    mensagem_alerta = Column(String(255), nullable=False)
    visualizado = Column(Boolean, default=False)
    ignorar_novos = Column(Boolean, default=False)
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\models\categoria.py

```
#models\categoria.py

from sqlalchemy import Column, Integer, String
from core.configs import settings

class Categoria(settings.DBBaseModel):
    __tablename__ = "categoria"

    categoria_id = Column(Integer, primary_key=True, index=True)
    nome_categoria = Column(String(50), nullable=False)  # Nome normalizado
    descricao_categoria = Column(String(255), nullable=True)
    nome_original = Column(String(100), nullable=False)  # Original do usuário
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\models\item.py

```
# models/item.py

from sqlalchemy import Column, Integer, String, Date, ForeignKey, TIMESTAMP, DateTime, Boolean
from datetime import datetime
from core.configs import settings

class Item(settings.DBBaseModel):
    __tablename__ = "item"

    item_id = Column(Integer, primary_key=True, index=True)
    nome_item = Column(String(256), nullable=False)
    descricao_item = Column(String(255), nullable=False)
    unidade_medida_item = Column(String(50), nullable=False)
    quantidade_item = Column(Integer, nullable=False)
    data_entrada_item = Column(DateTime, nullable=False, default=datetime.now)
    data_saida_item = Column(TIMESTAMP, nullable=True)
    data_validade_item = Column(Date, nullable=True)
    quantidade_minima_item = Column(Integer, nullable=True)
    categoria_id = Column(Integer, ForeignKey("categoria.categoria_id"), nullable=False)
    auditoria_usuario_id = Column(Integer, ForeignKey("usuario.usuario_id"), nullable=False)
    marca_item = Column(String(200), nullable=True)
```

```
    nome_item_original = Column(String(256), nullable=False)
    ativo = Column(Boolean, default=True, nullable=False) #  coluna para soft delete
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\models\retirada.py

```
#models\retirada.py

from sqlalchemy import Column, Integer, ForeignKey, String, DateTime, Text, Boolean
from sqlalchemy.orm import relationship
from core.configs import settings
from datetime import datetime
from enum import IntEnum

class StatusEnum(IntEnum):
    PENDENTE = 1
    AUTORIZADA = 2
    CONCLUIDA = 3
    NEGADA = 4


class Retirada(settings.DBBaseModel):
    __tablename__ = "retirada"

    retirada_id = Column(Integer, primary_key=True, index=True)
    usuario_id = Column(Integer, ForeignKey("usuario.usuario_id"), nullable=False)  # Quem solicitou
    autorizado_por = Column(Integer, ForeignKey("usuario.usuario_id"), nullable=True)  # Quem autorizou
    solicitado_localmente_por = Column(String(255), nullable=True)  # Nome de quem solicitou pessoalment
    setor_id = Column(Integer, ForeignKey("setor.setor_id"), nullable=False)
    status = Column(Integer, default=StatusEnum.PENDENTE, nullable=False)
    detalhe_status = Column(Text, nullable=True)  # Explicação do almoxarifado para autorização/negação
    justificativa = Column(Text, nullable=True)  # Justificativa do usuário
    data_solicitacao = Column(DateTime, default=datetime.now)
    is_active = Column(Boolean, default=True, nullable=False) # soft delete


    usuario = relationship("Usuario", foreign_keys=[usuario_id])
    admin = relationship("Usuario", foreign_keys=[autorizado_por])

    itens = relationship("RetiradaItem", back_populates="retirada")
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\models\retirada_item.py

```
#models\retirada_item.py

from sqlalchemy import Column, Integer, ForeignKey
from sqlalchemy.orm import relationship
from core.configs import settings

class RetiradaItem(settings.DBBaseModel):
    __tablename__ = "retirada_item"

    retirada_id = Column(Integer, ForeignKey("retirada.retirada_id"), primary_key=True)
    item_id = Column(Integer, ForeignKey("item.item_id"), primary_key=True)
    quantidade_retirada = Column(Integer, nullable=False)

    retirada = relationship("Retirada", back_populates="itens")
```

```
    item = relationship("Item")
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\models\setor.py

```
#models\setor.py

from sqlalchemy import Column, Integer, String
from core.configs import settings

class Setor(settings.DBBaseModel):
    __tablename__ = "setor"

    setor_id = Column(Integer, primary_key=True, index=True)
    nome_setor = Column(String(100), nullable=False)
    descricao_setor = Column(String(255), nullable=True)
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\models\usuario.py

```
#models\usuario.py

from sqlalchemy import Column, Integer, String, ForeignKey, Boolean
from core.configs import settings
from enum import IntEnum

class RoleEnum (IntEnum):
    USUARIO_GERAL = 1
    USUARIO_ALMOXARIFADO = 2
    USUARIO_DIRECAO = 3

class Usuario(settings.DBBaseModel):
    __tablename__ = "usuario"

    usuario_id = Column (Integer, primary_key=True, index=True)
    siape_usuario = Column (String(20), unique=True, nullable=True)
    nome_usuario = Column(String(100), nullable=False)
    tipo_usuario = Column (Integer, nullable=False)
    email_usuario = Column(String(100), unique=True, nullable=False)
    senha_usuario = Column(String(256), nullable=False)
    setor_id = Column (Integer, ForeignKey("setor.setor_id"), nullable=False)
    username = Column(String(100), nullable=False)
    is_active = Column (Boolean, nullable=False, default=True)
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\models__init__.py

```
#models\__init__.py

from core.configs import settings
from models.categoria import Categoria
from models.setor import Setor
from models.item import Item
from models.usuario import Usuario
from models.retirada import Retirada
```

```
from models.retirada_item import RetiradaItem
from models.alerta import Alerta
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\repositories\alerta_repository.py

```
# repositories/alerta_repository.py

from sqlalchemy import func, or_, and_ , update
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.future import select
from models.alerta import Alerta
from schemas.alerta import AlertaBase
from fastapi import HTTPException, status
from datetime import datetime

class AlertaRepository:

    @staticmethod
    async def alerta_ja_existe (db, tipo_alerta: int, item_id: int) -> bool:
        result = await db.execute(
            select(Alerta).where(
                Alerta.tipo_alerta == tipo_alerta,
                Alerta.item_id == item_id,

                # Um alerta é considerado "existente" e impede a criação de um novo se:
                or_(
                    Alerta.ignorar_novos == True, # Se foi marcado para ignorar novos alertas
                    and_(
                        Alerta.visualizado == False, # OU se não foi visualizado
                        Alerta.ignorar_novos == False # E não foi marcado para ignorar
                    )
                )
            )
        )
        return result.scalars().first() is not None

    @staticmethod
    async def create_alerta (db: AsyncSession, alerta_data: AlertaBase):
        novo_alerta = Alerta(
            tipo_alerta=alerta_data.tipo_alerta,
            item_id=alerta_data.item_id,
            mensagem_alerta=alerta_data.mensagem_alerta,
            data_alerta=datetime.now()
        )
        db.add(novo_alerta)
        await db.commit()
        return novo_alerta

    @staticmethod
    async def get_alertas (db: AsyncSession):
        result = await db.execute(select(Alerta))
        alertas = result.scalars().all()
        return alertas

    @staticmethod
    async def get_alerta_by_id(db: AsyncSession, alerta_id: int) -> Alerta | None:
        result = await db.execute(select(Alerta).where(Alerta.alerta_id == alerta_id))
        return result.scalars().first()

    @staticmethod
```

```
async def count_alertas(
    db: AsyncSession,
    tipo_alerta: int | None = None,
    search_term: str | None = None
) -> int:
    query = select(func.count()).select_from(Alerta)

    # Adicionar filtros
    if tipo_alerta is not None:
        query = query.where(Alerta.tipo_alerta == tipo_alerta)

    if search_term:
        # Tentar buscar por mensagem ou por ID do item.
        # Se a busca por item_id for numérica, precisa de tratamento para evitar erro
        try:
            item_id_int = int(search_term)
            query = query.where(
                (Alerta.mensagem_alerta.ilike(f"%{search_term}%")) |
                (Alerta.item_id == item_id_int)
            )
        except ValueError:
            # Se não for um ID de item numérico, busca apenas na mensagem
            query = query.where(Alerta.mensagem_alerta.ilike(f"%{search_term}%"))

    result = await db.execute(query)
    return result.scalar_one()

@staticmethod
async def get_alertas_paginated(
    db: AsyncSession,
    offset: int,
    limit: int,
    tipo_alerta: int | None = None,
    search_term: str | None = None
) -> list[Alerta]:
    query = select(Alerta)

    # Adicionar filtros (mesma lógica de count_alertas)
    if tipo_alerta is not None:
        query = query.where(Alerta.tipo_alerta == tipo_alerta)

    if search_term:
        try:
            item_id_int = int(search_term)
            query = query.where(
                (Alerta.mensagem_alerta.ilike(f"%{search_term}%")) |
                (Alerta.item_id == item_id_int)
            )
        except ValueError:
            query = query.where(Alerta.mensagem_alerta.ilike(f"%{search_term}%"))

    query = query.offset(offset).limit(limit).order_by(Alerta.data_alerta.desc())
    result = await db.execute(query)
    return result.scalars().all()

@staticmethod
async def delete_alerta (db: AsyncSession, alerta_id: int):
    alerta = await AlertaRepository.get_alerta_by_id(db, alerta_id)
    await db.delete(alerta)
    await db.commit()
    return {"message": "Alerta deletado com sucesso"}

@staticmethod
```

```
    async def ignorar_alerta (db: AsyncSession, alerta_id: int):
        alerta = await AlertaRepository.get_alerta_by_id(db, alerta_id)
        if alerta is None: # Adicionado para tratar o caso de alerta não encontrado
            return None
        alerta.ignorar_novos = True
        await db.commit()
        return alerta

    # Conta alertas não visualizados
    @staticmethod
    async def count_unviewed_alerts(db: AsyncSession) -> int:
        result = await db.execute(select(func.count()).select_from(Alerta).where(Alerta.visualizado == F
        return result.scalar_one()

    #Marca todos os alertas NÃO visualizados, como visualizados
    @staticmethod
    async def mark_all_alerts_as_viewed (db: AsyncSession):
        # Atualiza todos os alertas que NÃO foram visualizados para visualizado=True
        await db.execute(
            update(Alerta).
            where(Alerta.visualizado == False). # apenas alertas não visualizados
            values(visualizado=True) # definir visualizado como True
        )
        await db.commit()
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\repositories\categoria_repository.py

```
#repositories\categoria_repository.py
from sqlalchemy import func
from sqlalchemy.exc import NoResultFound, IntegrityError
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.future import select
from models.categoria import Categoria
from schemas.categoria import CategoriaCreate, CategoriaUpdate
from fastapi import HTTPException, status


class CategoriaRepository:

    @staticmethod
    async def create_categoria(db: AsyncSession, categoria_data: dict):
        nova_categoria = Categoria(**categoria_data)  # recebe um dicionário
        db.add(nova_categoria)
        await db.commit()
        await db.refresh(nova_categoria)
        return nova_categoria

    @staticmethod
    async def get_categorias(db: AsyncSession):
        result = await db.execute(select(Categoria))
        return result.scalars().all()

    @staticmethod
    async def get_categoria_by_id(db: AsyncSession, categoria_id: int):
        # Usa expressão SQLAlchemy para filtrar por ID
        return await CategoriaRepository.__first_or_404(
            db,
            Categoria.categoria_id == categoria_id
        )
    @staticmethod
```

```
async def aux_get_categoria_by_name(db: AsyncSession, categoria_name: str):
    result = await db.execute(
        select(Categoria)
        .where(Categoria.nome_categoria == categoria_name)
    )
    return result.scalars().first()  # Retorna None se não encontrar

@staticmethod
async def get_categoria_by_name(db: AsyncSession, categoria_name: str):
    # Filtra pelo nome exato
    return await CategoriaRepository.__first_or_404(
        db,
        Categoria.nome_categoria == categoria_name
    )

@staticmethod
async def get_categoria_by_name_like(db: AsyncSession, termo_busca: str):
    result = await db.execute(
        select(Categoria).where(
            Categoria.nome_categoria.ilike(f"%{termo_busca}%")
        )
    )
    return result

@staticmethod
async def update_categoria(db: AsyncSession, categoria_id: int, update_values: dict):
    categoria = await CategoriaRepository.__first_or_404(
        db,
        Categoria.categoria_id == categoria_id
    )

    for key, value in update_values.items():
        setattr(categoria, key, value)

    await db.commit()
    await db.refresh(categoria)
    return categoria

@classmethod
async def delete_categoria(cls, db: AsyncSession, categoria_id: int):
    expr = Categoria.categoria_id == categoria_id
    categoria = await cls.__first_or_404(db, expr)
    try:
        await db.delete(categoria)
        await db.commit()
        return {"message": "Categoria excluída com sucesso"}
    except IntegrityError as e:
        # se violação de FK em item → retorna 409 Conflict
        await db.rollback()
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail="Não é possível excluir: existem itens vinculados a esta categoria."
        )

@staticmethod
async def find_categoria_ids_by_name(db: AsyncSession, nome_normalizado: str) -> list[int]:
    result = await db.execute(
        select(Categoria.categoria_id)
        .where(Categoria.nome_categoria.ilike(f"%{nome_normalizado}%"))
    )
    return [r[0] for r in result.all()]

@staticmethod
```

```
async def count_categorias(db: AsyncSession) -> int:
    result = await db.execute(select(func.count()).select_from(Categoria))
    return result.scalar_one()

@staticmethod
async def get_categorias_paginated(
    db: AsyncSession,
    offset: int,
    limit: int
) -> list[Categoria]:
    result = await db.execute(
        select(Categoria)
        .offset(offset)
        .limit(limit)
    )
    return result.scalars().all()

@staticmethod
async def count_filtered_categorias(
    db: AsyncSession,
    categoria_ids: list[int] | None,
    nome_categoria_normalizado: str | None
) -> int:
    query = select(func.count()).select_from(Categoria)
    if categoria_ids:
        query = query.where(Categoria.categoria_id.in_(categoria_ids))
    if nome_categoria_normalizado:
        query = query.where(Categoria.nome_categoria.ilike(f"%{nome_categoria_normalizado}%"))
    result = await db.execute(query)
    return result.scalar_one()

@staticmethod
async def get_filtered_categorias_paginated(
    db: AsyncSession,
    categoria_ids: list[int] | None,
    nome_categorias_normalizado: str | None,
    offset: int, limit: int
) -> list[Categoria]:
    query = select(Categoria)
    if categoria_ids:
        query = query.where(Categoria.categoria_id.in_(categoria_ids))
    if nome_categorias_normalizado:
        query = query.where(Categoria.nome_categoria.ilike(f"%{nome_categorias_normalizado}%"))
    query = query.offset(offset).limit(limit)
    result = await db.execute(query)
    return result.scalars().all()

@staticmethod
async def __first_or_404(db: AsyncSession, where_expr):
    """
    Retorna o primeiro resultado para a expressão SQL where_expr,
    ou levanta HTTPException(404) se nenhum for encontrado.
    """
    try:
        result = await db.execute(
            select(Categoria).where(where_expr)
        )
        return result.scalars().one()
    except NoResultFound:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Categoria não encontrada"
        )
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\repositories\item_repository.py

```
# repositories/item_repository.py

from sqlalchemy import func, select
from sqlalchemy.ext.asyncio import AsyncSession
from models.item import Item
from models.categoria import Categoria

class ItemRepository:

    @staticmethod
    async def create(db: AsyncSession, item: Item) -> Item:
        db.add(item)
        await db.commit()
        await db.refresh(item)
        return item

    @staticmethod
    async def get_all(db: AsyncSession) -> list[Item]:
        # Retorna apenas itens ativos
        result = await db.execute(select(Item).where(Item.ativo == True))
        return result.scalars().all()

    @staticmethod
    async def get_by_id(db: AsyncSession, item_id: int) -> Item | None:
        # Retorna apenas itens ativos
        result = await db.execute(select(Item).where(Item.item_id == item_id, Item.ativo == True))
        return result.scalars().first()

    @staticmethod
    async def delete(db: AsyncSession, item: Item) -> None:
        # Soft delete: marca o item como inativo
        item.ativo = False
        await db.commit()
        await db.refresh(item) # Atualiza o objeto item no Python com o novo estado
        # Não é necessário db.delete(item) para soft delete

    @staticmethod
    async def count(db: AsyncSession) -> int:
        # Conta apenas itens ativos
        result = await db.execute(select(func.count()).select_from(Item).where(Item.ativo == True))
        return result.scalar_one()

    @staticmethod
    async def get_paginated(db: AsyncSession, offset: int, limit: int) -> list[Item]:
        # Retorna apenas itens ativos
        result = await db.execute(select(Item).where(Item.ativo == True).offset(offset).limit(limit))
        return result.scalars().all()

    @staticmethod
    async def find_filtered(
        db: AsyncSession,
        categoria_ids: list[int] | None = None,
        nome_produto_normalizado: str | None = None,
    ) -> list[tuple[Item, str]]:
        # Filtra apenas itens ativos
        query = select(Item, Categoria.nome_categoria).join(
            Categoria, Item.categoria_id == Categoria.categoria_id
```

```
    ).where(Item.ativo == True) # Adiciona filtro de ativo

    if categoria_ids:
        query = query.where(Item.categoria_id.in_(categoria_ids))

    if nome_produto_normalizado:
        query = query.where(
            Item.nome_item.ilike(f"%{nome_produto_normalizado}%")
        )

    result = await db.execute(query)
    return result.all()

@staticmethod
async def count_filtered(
    db: AsyncSession,
    categoria_ids: list[int] | None = None,
    nome_produto_normalizado: str | None = None,
) -> int:
    # Conta apenas itens ativos
    query = select(func.count()).select_from(Item).where(Item.ativo == True) # Adiciona filtro de at
    if categoria_ids:
        query = query.where(Item.categoria_id.in_(categoria_ids))

    if nome_produto_normalizado:
        query = query.where(
            Item.nome_item.ilike(f"%{nome_produto_normalizado}%")
        )

    result = await db.execute(query)
    return result.scalar_one()

@staticmethod
async def get_filtered_paginated(
    db: AsyncSession,
    categoria_ids: list[int] | None = None,
    nome_produto_normalizado: str | None = None,
    offset: int = 0,
    limit: int = 10,
) -> list[Item]:
    # Retorna apenas itens ativos
    query = select(Item).where(Item.ativo == True) # Adiciona filtro de ativo

    if categoria_ids:
        query = query.where(Item.categoria_id.in_(categoria_ids))

    if nome_produto_normalizado:
        query = query.where(
            Item.nome_item.ilike(f"%{nome_produto_normalizado}%")
        )

    query = query.offset(offset).limit(limit)
    result = await db.execute(query)
    return result.scalars().all()

@staticmethod
async def get_items_by_category(db: AsyncSession, categoria_id: int) -> list[Item]:
    # Retorna apenas itens ativos
    result = await db.execute(select(Item).where(Item.categoria_id == categoria_id, Item.ativo == Tr
    return result.scalars().all()

@staticmethod
async def get_items_period(
```

```
        db: AsyncSession, data_inicio, data_fim
    ) -> list[dict]:
        # Retorna apenas itens ativos
        query = (
            select(
                Item.item_id,
                Item.nome_item_original,
                Item.quantidade_item,
                Item.data_entrada_item,
                Categoria.nome_original.label("nome_categoria_original"),
            )
            .join(Categoria, Item.categoria_id == Categoria.categoria_id)
            .where(
                Item.data_entrada_item >= data_inicio,
                Item.data_entrada_item <= data_fim,
                Item.ativo == True # Adiciona filtro de ativo
            )
        )
        result = await db.execute(query)
        return result.mappings().all()

    @staticmethod
    async def get_items_expiring_before(db: AsyncSession, date) -> list[Item]:
        # Retorna apenas itens ativos
        result = await db.execute(
            select(Item).where(Item.data_validade_item <= date, Item.ativo == True) # Adiciona filtro de
        )
        return result.scalars().all()

    @staticmethod
    async def find_low_stock(db: AsyncSession) -> list[Item]:
        # Retorna apenas itens ativos
        result = await db.execute(
            select(Item).where(Item.quantidade_item < Item.quantidade_minima_item, Item.ativo == True) #
        )
        return result.scalars().all()
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\repositories\retirada_repository.py

```
# repositories/retirada_repository.py

from sqlalchemy import func, and_, or_, update # Importe 'update'
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.future import select
from sqlalchemy.orm import selectinload, aliased
from schemas.retirada import RetiradaFilterParams
from models.retirada import Retirada, StatusEnum
from models.retirada_item import RetiradaItem
from models.item import Item
from models.usuario import Usuario
from datetime import datetime

class RetiradaRepository:

    @staticmethod
    async def count_retiradas(db: AsyncSession) -> int:
        """Conta o total de retiradas ativas no banco de dados."""
        result = await db.execute(select(func.count(Retirada.retirada_id)).where(Retirada.is_active == T
        return result.scalar_one()
    @staticmethod
```

```
async def get_retiradas_paginated(db: AsyncSession, offset: int, limit: int):
    """Retorna uma lista paginada de retiradas ativas, ordenadas por data_solicitacao (desc),
    com eager loading de itens, usuário e admin."""
    q = (
        select(Retirada)
        .options(
            selectinload(Retirada.itens).selectinload(RetiradaItem.item),
            selectinload(Retirada.usuario),
            selectinload(Retirada.admin),
        )
        .where(Retirada.is_active == True) # Filtra apenas retiradas ativas
        .order_by(Retirada.data_solicitacao.desc()) # ordena do mais recente para o mais antigo
        .offset(offset)
        .limit(limit)
    )
    return (await db.execute(q)).scalars().all()

@staticmethod
async def count_retiradas_pendentes(db: AsyncSession) -> int:
    """Conta o total de retiradas ativas com status PENDENTE."""
    result = await db.execute(
        select(func.count(Retirada.retirada_id))
        .where(and_(Retirada.status == StatusEnum.PENDENTE, Retirada.is_active == True)) # Adiciona
    )
    return result.scalar_one()

@staticmethod
async def get_retiradas_pendentes_paginated(db: AsyncSession, offset: int, limit: int):
    """Retorna uma lista paginada de retiradas ativas e pendentes, ordenadas por data_solicitacao (d
    com eager loading de itens, usuário e admin."""
    q = (
        select(Retirada)
        .options(
            selectinload(Retirada.itens).selectinload(RetiradaItem.item),
            selectinload(Retirada.usuario),
            selectinload(Retirada.admin),
        )
        .where(and_(Retirada.status == StatusEnum.PENDENTE, Retirada.is_active == True)) # Adiciona
        .order_by(Retirada.data_solicitacao.desc()) # pendentes mais recentes primeiro
        .offset(offset)
        .limit(limit)
    )
    return (await db.execute(q)).scalars().all()

@staticmethod
async def count_retiradas_filter(db: AsyncSession, params: RetiradaFilterParams) -> int:
    """Conta o total de retiradas ativas filtradas com base nos parâmetros fornecidos."""
    q = select(func.count(Retirada.retirada_id)).where(Retirada.is_active == True) # Adiciona filtro
    conditions = []

    if params.status is not None:
        conditions.append(Retirada.status == params.status)
    if params.solicitante:
        alias = aliased(Usuario)
        q = q.select_from(Retirada).join(alias, Retirada.usuario)
        conditions.append(
            or_(
                alias.nome_usuario.ilike(f"%{params.solicitante}%"),
                Retirada.solicitado_localmente_por.ilike(f"%{params.solicitante}%")
            )
        )
    if params.start_date and params.end_date:
        conditions.append(
```

```
            and_(
                Retirada.data_solicitacao >= params.start_date, # Usar >=
                Retirada.data_solicitacao <= params.end_date # Usar <=
            )
        )
    if conditions:
        q = q.where(and_(*conditions)) # Usa and_ para combinar todas as condições

    return (await db.execute(q)).scalar_one()

@staticmethod
async def filter_retiradas_paginated(
    db: AsyncSession,
    params: RetiradaFilterParams,
    offset: int,
    limit: int
):
    """Filtra e retorna retiradas ativas paginadas com base nos parâmetros fornecidos,
    ordenadas por data_solicitacao (desc), com eager loading de itens, usuário e admin."""
    q = select(Retirada).where(Retirada.is_active == True) # Adiciona filtro de ativo
    q = q.options(
        selectinload(Retirada.itens).selectinload(RetiradaItem.item),
        selectinload(Retirada.usuario),
        selectinload(Retirada.admin),
    )

    conditions = []
    if params.status is not None:
        conditions.append(Retirada.status == params.status)

    if params.solicitante:
        alias = aliased(Usuario)
        q = q.join(alias, Retirada.usuario) # Join aqui para usar o alias
        conditions.append(
            or_(
                alias.nome_usuario.ilike(f"%{params.solicitante}%"),
                Retirada.solicitado_localmente_por.ilike(f"%{params.solicitante}%")
            )
        )
    if params.start_date and params.end_date:
        conditions.append(
            and_(
                Retirada.data_solicitacao >= params.start_date, # Usar >=
                Retirada.data_solicitacao <= params.end_date # Usar <=
            )
        )
    if conditions:
        q = q.where(and_(*conditions))

    q = (
        q.order_by(Retirada.data_solicitacao.desc())
        .offset(offset)
        .limit(limit)
    )
    return (await db.execute(q)).scalars().all()

@staticmethod
async def criar_retirada(db: AsyncSession, retirada: Retirada):
    """Adiciona uma nova retirada ao banco de dados e a atualiza para obter o ID."""
    db.add(retirada)
    await db.flush()
    await db.refresh(retirada)
    return retirada
```

```python
@staticmethod
async def adicionar_itens_retirada(db: AsyncSession, itens: list[RetiradaItem]):
    """Adiciona múltiplos itens a uma retirada no banco de dados."""
    db.add_all(itens)
    await db.flush()

@staticmethod
async def buscar_retirada_por_id(db: AsyncSession, retirada_id: int):
    """Busca uma retirada ativa específica pelo ID, com eager loading de itens, usuário e admin."""
    result = await db.execute(
        select(Retirada)
        .options(
            selectinload(Retirada.itens).selectinload(RetiradaItem.item),
            selectinload(Retirada.usuario),
            selectinload(Retirada.admin),
        )
        .where(and_(Retirada.retirada_id == retirada_id, Retirada.is_active == True)) # Adiciona fil
    )
    return result.scalars().first()

@staticmethod
async def get_retiradas(db: AsyncSession):
    """Retorna todas as retiradas ativas (sem paginação), ordenadas por data_solicitacao (desc), com
    result = await db.execute(
        select(Retirada)
        .options(
            selectinload(Retirada.itens).selectinload(RetiradaItem.item),
            selectinload(Retirada.usuario),
            selectinload(Retirada.admin),
        )
        .where(Retirada.is_active == True) # Adiciona filtro de ativo
        .order_by(Retirada.data_solicitacao.desc()) # ordena do mais recente para o mais antigo
    )
    return result.scalars().unique().all()

@staticmethod
async def get_retiradas_por_setor_periodo(
    db: AsyncSession,
    setor_id: int,
    data_inicio: datetime,
    data_fim: datetime
):
    """Retorna retiradas ativas filtradas por setor e período, com eager loading."""
    result = await db.execute(
        select(Retirada)
        .options(
            selectinload(Retirada.itens).selectinload(RetiradaItem.item),
            selectinload(Retirada.usuario),
            selectinload(Retirada.admin),
        )
        .where(
            Retirada.setor_id == setor_id,
            Retirada.data_solicitacao >= data_inicio, # Usar >=
            Retirada.data_solicitacao <= data_fim, # Usar <=
            Retirada.is_active == True # Adiciona filtro de ativo
        )
        .order_by(Retirada.data_solicitacao.desc())
    )
    return result.scalars().unique().all()

@staticmethod
async def get_retiradas_por_usuario_periodo(
    db: AsyncSession,
```

```
    usuario_id: int,
    data_inicio: datetime,
    data_fim: datetime
):
    """Retorna retiradas ativas filtradas por usuário e período, com eager loading."""
    result = await db.execute(
        select(Retirada)
        .options(
            selectinload(Retirada.itens).selectinload(RetiradaItem.item),
            selectinload(Retirada.usuario),
            selectinload(Retirada.admin),
        )
        .where(
            Retirada.usuario_id == usuario_id,
            Retirada.data_solicitacao >= data_inicio, # Usar >=
            Retirada.data_solicitacao <= data_fim, # Usar <=
            Retirada.is_active == True # Adiciona filtro de ativo
        )
        .order_by(Retirada.data_solicitacao.desc())
    )
    return result.scalars().unique().all()

@staticmethod
async def atualizar_retirada(db: AsyncSession, retirada: Retirada):
    """Atualiza uma retirada existente no banco de dados."""
    # Não é necessário filtrar por is_active aqui, pois o objeto `retirada` já foi buscado
    # e estamos atualizando ele.
    await db.commit()
    await db.refresh(retirada)
    return retirada

@staticmethod
async def buscar_item_por_id(db: AsyncSession, item_id: int):
    """Busca um item pelo ID."""
    result = await db.execute(select(Item).where(Item.item_id == item_id))
    return result.scalars().first()

@staticmethod
async def atualizar_quantidade_item(db: AsyncSession, item: Item, nova_quantidade: int):
    """Atualiza a quantidade de um item no estoque."""
    item.quantidade_item = nova_quantidade
    await db.flush()
    return item

@staticmethod
async def get_retiradas_by_user_paginated(db: AsyncSession, usuario_id: int, offset: int, limit: int
    """Retorna retiradas ativas paginadas para um usuário específico, ordenadas por data_solicitacao
    com eager loading de itens, usuário e admin."""
    query = (
        select(Retirada)
        .where(and_(Retirada.usuario_id == usuario_id, Retirada.is_active == True)) # Adiciona filtr
        .options(
            selectinload(Retirada.itens).selectinload(RetiradaItem.item),
            selectinload(Retirada.usuario),
            selectinload(Retirada.admin),
        )
        .order_by(Retirada.data_solicitacao.desc())
        .offset(offset)
        .limit(limit)
    )
    result = await db.execute(query)
    return result.scalars().all()
@staticmethod
```

```
    async def count_retiradas_by_user(db: AsyncSession, usuario_id: int) -> int:
        """Conta o total de retiradas ativas para um usuário específico."""
        result = await db.execute(
            select(func.count(Retirada.retirada_id))
            .where(and_(Retirada.usuario_id == usuario_id, Retirada.is_active == True)) # Adiciona filtr
        )
        return result.scalar_one()

    @staticmethod
    async def soft_delete_by_period(db: AsyncSession, start_date: datetime, end_date: datetime) -> int:
        """
        Marca retiradas como inativas (soft delete) dentro de um período de datas.
        Retorna o número de retiradas atualizadas.
        """
        # Garante que apenas retiradas ativas sejam "deletadas" e que elas realmente estejam no período
        result = await db.execute(
            update(Retirada)
            .where(
                Retirada.is_active == True,
                Retirada.data_solicitacao >= start_date,
                Retirada.data_solicitacao <= end_date
            )
            .values(is_active=False)
        )
        await db.commit()
        return result.rowcount # Retorna o número de linhas afetadas
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\repositories\setor_repository.py

```
# repositories/setor_repository.py
from sqlalchemy.orm import Session
from models.setor import Setor
from schemas.setor import SetorCreate, SetorUpdate
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.future import select
from fastapi import HTTPException, status

class SetorRepository:

    #persistir um novo setor na base de dados
    @staticmethod
    async def create_setor(db: AsyncSession, setor_data: SetorCreate):
        novo_setor = Setor(
            nome_setor=setor_data.nome_setor,
            descricao_setor=setor_data.descricao_setor
        )
        db.add(novo_setor)
        await db.commit()
        await db.refresh(novo_setor)
        return novo_setor

    #listar todos os setores da base de dados
    @staticmethod
    async def get_setores(db: Session):
        result = await db.execute(select(Setor))
        setores = result.scalars().all()
        return setores

    #filtrar um setor por id
    @staticmethod
```

```
    async def get_setor_by_id(db: Session, setor_id: int):
        result = await db.execute(select(Setor).filter(Setor.setor_id == setor_id))
        setor = result.scalars().first()

        if not setor:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Setor não encontrado.")

        return setor

    #atualizar dados do setor por id
    @staticmethod
    async def update_setor(db: AsyncSession, setor_id: int, setor_data: SetorUpdate):
        # Busca o setor pelo ID
        result = await db.execute(select(Setor).filter(Setor.setor_id == setor_id))
        setor = result.scalars().first()

        if not setor:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Setor não encontrado.")

        # Atualiza os campos do setor
        setor.nome_setor = setor_data.nome_setor
        setor.descricao_setor = setor_data.descricao_setor

        # Persiste as alterações no banco de dados
        await db.commit()
        await db.refresh(setor)

        return setor

    #deletar setor por id
    @staticmethod
    async def delete_setor(db: AsyncSession, setor_id: int):
        # Busca o setor pelo ID
        result = await db.execute(select(Setor).filter(Setor.setor_id == setor_id))
        setor = result.scalars().first()

        if not setor:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Setor não encontrado.")

        # Remove o setor do banco de dados
        await db.delete(setor)
        await db.commit()

        return setor
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\repositories\usuario_repository.py

```
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.future import select
from models.usuario import Usuario
from schemas.usuario import UsuarioCreate, UsuarioUpdate
from core.security import get_password_hash
from fastapi import HTTPException, status

class UsuarioRepository:
    @staticmethod
    async def create_usuario(db: AsyncSession, user_data: UsuarioCreate):

        #criando o modelo de usuário
        new_user = Usuario(
```

```
        siape_usuario=user_data.siape_usuario,
        nome_usuario=user_data.nome_usuario,
        senha_usuario=get_password_hash(user_data.senha_usuario),
        tipo_usuario=user_data.tipo_usuario,
        email_usuario=user_data.email_usuario,
        setor_id=user_data.setor_id,
        username=user_data.username,

    )

    db.add(new_user)
    await db.commit()
    await db.refresh(new_user)
    return new_user



@staticmethod
async def get_usuarios(db: AsyncSession):
    result = await db.execute(
        select(Usuario).where(Usuario.is_active == True)
    )
    return result.scalars().all()

@staticmethod
async def get_usuario_by_id(db: AsyncSession, usuario_id: int):
    result = await db.execute(
        select(Usuario)
        .where(Usuario.usuario_id == usuario_id, Usuario.is_active == True)
    )
    usuario = result.scalars().first()
    if not usuario:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Usuário não encontrado ou inativo"
        )
    return usuario

#deletar usuário (soft delete)
@staticmethod
async def delete_usuario(db: AsyncSession, usuario_id: int):
    result = await db.execute(
        select(Usuario).where(Usuario.usuario_id == usuario_id)
    )
    usuario = result.scalars().first()
    if not usuario:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Usuário não encontrado"
        )
    # Soft delete:
    usuario.is_active = False
    await db.commit()
    return {"message": "Usuário inativado com sucesso"}

# atualizar dados do usuario
@staticmethod
async def update_usuario(db: AsyncSession, usuario_id: int, usuario_data: UsuarioUpdate):
    result = await db.execute(select(Usuario).where(Usuario.usuario_id == usuario_id))
    usuario = result.scalars().first()

    if not usuario:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Usuário não encontrado")
```

```
        # Atualiza apenas os campos enviados na requisição
        if usuario_data.nome_usuario:
            usuario.nome_usuario = usuario_data.nome_usuario

        if usuario_data.email_usuario:
            usuario.email_usuario = usuario_data.email_usuario.lower()

        if usuario_data.tipo_usuario is not None:
            usuario.tipo_usuario = usuario_data.tipo_usuario

        if usuario_data.setor_id is not None:
            usuario.setor_id = usuario_data.setor_id

        if usuario_data.senha_usuario:
            usuario.senha_usuario = get_password_hash(usuario_data.senha_usuario)

        if usuario_data.username:
            usuario.username = usuario_data.username.lower()

        await db.commit()
        await db.refresh(usuario)
        return usuario
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\repositories__init__.py

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\schemas\alerta.py

```
from pydantic import BaseModel
from datetime import datetime
from typing import List, Optional

class AlertaBase (BaseModel):
    tipo_alerta: int
    item_id: int
    data_alerta: datetime
    mensagem_alerta: str
    visualizado: bool = False
    ignorar_novos: bool = False

class AlertaOut (AlertaBase):
    alerta_id: int

    class Config:
        from_attributes = True

# schema para paginação
class PaginatedAlertas(BaseModel):
    page: int
    size: int
    total: int # Total de alertas no banco
    total_pages: int # Total de páginas
    items: List[AlertaOut]

    model_config = {
        'from_attributes': True
```

```
    }
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\schemas\auth_schemas.py

```
from pydantic import BaseModel

class TokenSchema(BaseModel):
    access_token: str
    token_type: str
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\schemas\categoria.py

```
# schemas/categoria.py
from pydantic import BaseModel
from typing import List, Optional

class CategoriaBase(BaseModel):
    nome_categoria: str
    descricao_categoria: Optional[str] = None

class CategoriaCreate(CategoriaBase):
    pass

class CategoriaOut(CategoriaBase):
    categoria_id: int
    nome_original: str  # Campo para armazenar o nome original (derivado de nome_categoria)
    nome_categoria: str  # Campo normalizado

    class Config:
        from_attributes = True

class CategoriaUpdate(BaseModel):
    nome_categoria: Optional[str] = None
    descricao_categoria: Optional[str] = None

class PaginatedCategorias(BaseModel):
    page: int
    size: int
    total: int           # total de categorias no banco
    total_pages: int     # total de páginas (ceil(total/size))
    items: List[CategoriaOut]

    model_config = {
        'from_attributes': True
    }
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\schemas\item.py

```
# schemas/item.py

from pydantic import BaseModel, Field
from datetime import date, datetime
```

```
from typing import List, Optional

class ItemBase(BaseModel):
    nome_item: str
    descricao_item: str
    quantidade_item: int
    categoria_id: int
    data_validade_item: Optional[date] = None
    quantidade_minima_item: Optional[int] = None
    marca_item: Optional[str] = None

class ItemCreate(BaseModel):
    nome_item: str
    descricao_item: str
    unidade_medida_item: str
    quantidade_item: int
    categoria_id: int
    data_validade_item: Optional[datetime] = None
    quantidade_minima_item: Optional[int] = None
    data_entrada_item: Optional[datetime] = None
    marca_item: Optional[str] = None

class ItemUpdate(BaseModel):
    nome_item: Optional[str] = None
    descricao_item: Optional[str] = None
    quantidade_item: Optional[int] = None
    categoria_id: Optional[int] = None
    data_validade_item: Optional[date] = None
    data_entrada_item: Optional[datetime] = None
    data_saida_item: Optional[datetime] = None
    quantidade_minima_item: Optional[int] = None
    marca_item: Optional[str] = None
    unidade_medida_item: Optional[str] = None
    ativo: Optional[bool] = None #  para permitir atualização do status ativo/inativo

class ItemOut(ItemBase):
    item_id: int
    nome_item: str # Nome normalizado Apenas para lógica interna
    descricao_item: str
    quantidade_item: int
    categoria_id: int
    data_validade_item: Optional[date] = None
    auditoria_usuario_id: int
    quantidade_minima_item: Optional[int] = None
    marca_item: Optional[str] = None
    unidade_medida_item: str
    data_entrada_item: Optional[datetime] = None
    nome_item_original: str # nome original conforme enviado pelo usuário Exibição no front
    ativo: bool #  para soft delete

    class Config:
        from_attributes = True

class PaginatedItems(BaseModel):
    page: int
    size: int
    total: int # total de itens no banco
    total_pages: int # total de páginas (ceil(total/size))
    items: List[ItemOut]

    model_config = {
        'from_attributes': True
    }
```

```
# Schema para o resultado do upload em massa
class BulkItemUploadResult(BaseModel):
    total_items_processed: int
    items_created: int
    items_updated: int
    errors: List[dict] = [] # Lista de dicionários com {"row": <número da linha>, "error": <mensagem de

    model_config = {
        'from_attributes': True
    }
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\schemas\retirada.py

```
# schemas/retirada.py
from pydantic import BaseModel
from typing import Optional, List
from datetime import datetime
from models.retirada import StatusEnum
from schemas.item import ItemOut

class ItemRetirada(BaseModel):
    item_id: int
    quantidade_retirada: int

class RetiradaBase(BaseModel):
    setor_id: int
    justificativa: Optional[str] = None
    solicitado_localmente_por: Optional[str] = None

class RetiradaCreate(RetiradaBase):
    itens: List[ItemRetirada]

class RetiradaUpdateStatus(BaseModel):
    status: StatusEnum
    detalhe_status: Optional[str] = None

class RetiradaItemOut(BaseModel):
    item_id: int
    quantidade_retirada: int
    item: ItemOut

    model_config = {"from_attributes": True}

class RetiradaOut(BaseModel):
    retirada_id: int
    usuario_id: int
    autorizado_por: Optional[int]
    setor_id: int
    status: StatusEnum
    detalhe_status: Optional[str]
    justificativa: Optional[str]
    solicitado_localmente_por: Optional[str]
    data_solicitacao: datetime
    itens: List[RetiradaItemOut]

    model_config = {"from_attributes": True}

class RetiradaPaginated(BaseModel):
    total: int
    page: int
```

```
    pages: int
    items: List[RetiradaOut]

    model_config = {"from_attributes": True}

class RetiradaFilterParams(BaseModel):
    status: Optional[int] = None
    solicitante: Optional[str] = None
    start_date: Optional[datetime] = None
    end_date: Optional[datetime] = None
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\schemas\setor.py

```
# schemas/setor.py
from pydantic import BaseModel
from typing import Optional

class SetorBase(BaseModel):
    nome_setor: str
    descricao_setor: Optional[str] = None

class SetorCreate(SetorBase):
    pass

class SetorUpdate(SetorBase):
    pass

class SetorOut(SetorBase):
    setor_id: int

    class Config:
        from_attributes = True
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\schemas\usuario.py

```
#schemas/usuario.py
from pydantic import BaseModel, EmailStr
from typing import Optional

class UsuarioBase (BaseModel):
    nome_usuario: str
    email_usuario: EmailStr
    tipo_usuario: int
    setor_id: int
    username: str
    siape_usuario: Optional[str] = None

class UsuarioCreate (UsuarioBase):
    nome_usuario: str
    siape_usuario: Optional [str] = None
    tipo_usuario: int
    senha_usuario: str
    email_usuario: EmailStr
    setor_id: int
    username: str
class UsuarioUpdate (BaseModel):
```

```
    nome_usuario: Optional [str] = None
    siape_usuario: Optional[str] = None
    tipo_usuario: Optional [int] = None
    senha_usuario: Optional [str] = None
    email_usuario: Optional [EmailStr] = None
    setor_id: Optional [int] = None
    username: Optional [str] = None

class UsuarioOut (UsuarioBase):
    usuario_id: int

    class Config:
        from_attributes = True

# SCHEMA PARA REDEFINIÇÃO DE SENHA SIMPLES
class UsuarioResetPasswordSimple(BaseModel):
    username_or_email: str
    new_password: str

# SCHEMA PARA CHECAGEM DE USUÁRIO PARA REDEFINIÇÃO DE SENHA
class UsuarioCheckForReset(BaseModel):
    username_or_email: str
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\schemas__init__.py

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\alerta_service.py

```
# services/alerta_service.py
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy import select
from datetime import datetime, timedelta
from models.alerta import TipoAlerta
from models.item import Item
from repositories.item_repository import ItemRepository
from repositories.alerta_repository import AlertaRepository
from schemas.alerta import AlertaBase, PaginatedAlertas, AlertaOut
from utils.websocket_endpoints import manager # Importar o manager
from fastapi import HTTPException, status
import math

class AlertaService:
    @staticmethod
    async def generate_daily_alerts(db: AsyncSession):
        # Verificar validade e estoque juntos
        await AlertaService.verificar_validade_itens(db)
        await AlertaService.verificar_estoque_baixo(db)

    @staticmethod
    async def verificar_validade_itens(db: AsyncSession):
        threshold_date = datetime.now() + timedelta(days=60)
        items = await ItemRepository.get_items_expiring_before(db, threshold_date)

        for item in items:
            alerta_existe = await AlertaRepository.alerta_ja_existe(
                db, TipoAlerta.VALIDADE_PROXIMA.value, item.item_id
```

```
        )
        if not alerta_existe:
            # Criar o alerta
            novo_alerta = await AlertaRepository.create_alerta(db, AlertaBase(
                tipo_alerta=TipoAlerta.VALIDADE_PROXIMA.value,
                mensagem_alerta=f"Item {item.nome_item_original} próximo da validade",
                item_id=item.item_id,
                data_alerta=datetime.now()
            ))
            #  Transmitir o evento de novo alerta via WebSocket (para conexões gerais)
            await manager.broadcast({"type": "new_alert", "alert_id": novo_alerta.alerta_id, "messag

@staticmethod
async def get_alertas(db: AsyncSession):
    result = await AlertaRepository.get_alertas(db)
    if not result:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail="Não foram encontrados alertas na base de dados")
    return result

@staticmethod
async def verificar_estoque_baixo(db: AsyncSession, item_id: int = None):
    query = select(Item)
    if item_id:
        query = query.where(Item.item_id == item_id)
    query = query.where(Item.quantidade_item < Item.quantidade_minima_item)

    items = await db.execute(query)

    for item in items.scalars():
        alerta_existe = await AlertaRepository.alerta_ja_existe(
            db, TipoAlerta.ESTOQUE_BAIXO.value, item.item_id
        )
        if not alerta_existe:
            # Criar o alerta
            novo_alerta = await AlertaRepository.create_alerta(db, AlertaBase(
                tipo_alerta=TipoAlerta.ESTOQUE_BAIXO.value,
                mensagem_alerta=f"Estoque de {item.nome_item_original} abaixo do mínimo",
                item_id=item.item_id,
                data_alerta=datetime.now()
            ))
            # Transmitir o evento de novo alerta via WebSocket (para conexões gerais)
            await manager.broadcast({"type": "new_alert", "alert_id": novo_alerta.alerta_id, "messag

@staticmethod
async def get_alerta_by_id(db: AsyncSession, alerta_id: int):
    result = await AlertaRepository.get_alerta_by_id(db, alerta_id)
    if not result:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Alerta não encontrado")
    return result

@staticmethod
async def get_alertas_paginated(
    db: AsyncSession,
    page: int,
    size: int,
    tipo_alerta: int = None,
    search_term: str = None
) -> PaginatedAlertas:
    allowed_sizes = [5,10,25,50,100]
    if size not in allowed_sizes:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
```

```
                detail=f"size deve ser um de {allowed_sizes}"
            )
        if page < 1:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="page deve ser >= 1"
            )

        total_alertas = await AlertaRepository.count_alertas(db, tipo_alerta, search_term) # Passar fil
        total_pages = math.ceil(total_alertas / size) if total_alertas > 0 else 1
        offset = (page - 1) * size
        alertas_db = await AlertaRepository.get_alertas_paginated(db, offset, size, tipo_alerta, search_
        items_out = [AlertaOut.model_validate(alerta) for alerta in alertas_db]

        return PaginatedAlertas(
            page=page,
            size=size,
            total=total_alertas,
            total_pages=total_pages,
            items=items_out
        )

    @staticmethod
    async def mark_alerta_as_ignorar_novos(db: AsyncSession, alerta_id: int):
        alerta = await AlertaRepository.ignorar_alerta(db, alerta_id)
        if not alerta:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Alerta não encontrado.")
        return alerta

    @staticmethod
    async def get_unviewed_alerts_count(db: AsyncSession) -> int:
        return await AlertaRepository.count_unviewed_alerts(db)

    @staticmethod
    async def mark_all_alerts_as_viewed(db: AsyncSession):
        await AlertaRepository.mark_all_alerts_as_viewed(db)

    @staticmethod
    async def delete_alerta(db: AsyncSession, alerta_id: int):
        alerta = await AlertaRepository.get_alerta_by_id(db, alerta_id)
        if not alerta:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Alerta não encontrado.")
        await AlertaRepository.delete_alerta(db, alerta_id)
        return {"message": "Alerta deletado com sucesso"}
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\bulk_processor.py

```
# app/services/item/bulk_processor.py

import io
from datetime import datetime
from fastapi import HTTPException, status
from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession
import pandas as pd

from utils.normalizar_texto import normalize_name
from services.finder import ItemFinder
from repositories.item_repository import ItemRepository
from schemas.item import BulkItemUploadResult
```

```
# Import do modelo Categoria para criação de novas categorias
from models.categoria import Categoria


class ItemBulkProcessor:
    ALLOWED_CONTENT_TYPES = {
        "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet": "xlsx",
        "text/csv": "csv",
    }

    def __init__(self, db: AsyncSession, auditoria_usuario_id: int):
        self.db = db
        self.auditoria_usuario_id = auditoria_usuario_id
        # Mapeamento de nome_categoria_normalizado -> categoria_id
        self.category_map: dict[str, int] = {}

    async def process(self, upload_file) -> BulkItemUploadResult:
        content_type = upload_file.content_type
        if content_type not in self.ALLOWED_CONTENT_TYPES:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="Tipo de arquivo inválido. Apenas .xlsx e .csv são permitidos.",
            )

        # 1) Ler DataFrame conforme tipo
        df = await self._load_dataframe(upload_file)

        # 2) Preparar colunas e pré-carregar/criar categorias
        df = self._normalize_columns(df)
        await self._fetch_or_create_categories(df)

        total_processed = len(df)
        self._items_created = 0
        self._items_updated = 0
        errors: list[dict] = []

        # 3) Processar cada linha separadamente
        for idx, row in df.iterrows():
            try:
                await self._process_row(idx, row)
            except ValueError as ve:
                errors.append({"row": idx + 2, "error": str(ve)})
                await self.db.rollback()
            except Exception as e:
                errors.append({"row": idx + 2, "error": f"Erro inesperado: {e}"})
                await self.db.rollback()

        # 4) Commit final (se ao menos uma linha foi processada sem erro crítico)
        await self.db.commit()

        return BulkItemUploadResult(
            total_items_processed=total_processed,
            items_created=self._items_created,
            items_updated=self._items_updated,
            errors=errors,
        )

    async def _load_dataframe(self, upload_file) -> pd.DataFrame:
        content = await upload_file.read()
        file_type = self.ALLOWED_CONTENT_TYPES[upload_file.content_type]

        if file_type == "xlsx":
            return pd.read_excel(io.BytesIO(content))
```

```
    else:  # csv
        text = content.decode("utf-8")
        return pd.read_csv(io.StringIO(text))

def _normalize_columns(self, df: pd.DataFrame) -> pd.DataFrame:
    df.columns = df.columns.str.strip().str.lower()
    required = ["produto", "quantidade", "unidade de medida", "categoria"]
    for col in required:
        if col not in df.columns:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail=f"Coluna obrigatória '{col}' não encontrada no arquivo.",
            )
    # 'descrição' é opcional agora
    return df

async def _fetch_or_create_categories(self, df: pd.DataFrame) -> None:
    """
    1) Extrai todas as categorias normalizadas do DataFrame.
    2) Busca as que já existem no banco de dados e atualiza self.category_map.
    3) Para as categorias inexistentes, cria novas instâncias de Categoria,
       persiste no banco (flush) e inclui no self.category_map.
    """
    # 1) Normaliza nomes de categoria únicos do arquivo
    raw_categories = df["categoria"].dropna().astype(str).str.strip().tolist()
    normalized_list = [normalize_name(name) for name in raw_categories]
    unique_normalized = list(dict.fromkeys(normalized_list))  # mantém ordem, sem duplicar

    # 2) Busca no banco as categorias que já existem
    query = select(Categoria).where(Categoria.nome_categoria.in_(unique_normalized))
    result = await self.db.execute(query)
    existing_categories = result.scalars().all()

    # Preenche o mapeamento para as já existentes
    for cat in existing_categories:
        self.category_map[cat.nome_categoria] = cat.categoria_id

    # 3) Identifica quais normalized names ainda não existem e cria
    existing_names = {cat.nome_categoria for cat in existing_categories}
    to_create = [nome for nome in unique_normalized if nome not in existing_names]

    for normalized_cat in to_create:
        # Descobrir o raw name correspondente (para nome_original)
        raw_index = normalized_list.index(normalized_cat)
        raw_name = raw_categories[raw_index]

        nova_categoria = Categoria(
            nome_original=raw_name,
            nome_categoria=normalized_cat,
        )
        self.db.add(nova_categoria)
        await self.db.flush()  # garante que nova_categoria.categoria_id seja populado

        # Atualiza o mapeamento
        self.category_map[normalized_cat] = nova_categoria.categoria_id

async def _process_row(self, idx: int, row) -> None:
    """
    Processa uma linha do DataFrame:
     - Valida campos obrigatórios de cada coluna
     - Normaliza nome do item e procura duplicata
     - Se existe duplicata: atualiza quantidade + campos opcionais
     - Senão: cria novo objeto Item e dá db.add(...)
```

```
 - Usa nome_item + unidade como 'descrição' quando coluna ausente ou vazia
"""
# --- 1. Extrair e validar campos básicos ---
produto_raw = str(row["produto"]).strip()
if not produto_raw:
    raise ValueError("Nome do produto não pode ser vazio.")

qtd_raw = row["quantidade"]
if pd.isna(qtd_raw) or not str(qtd_raw).strip().isdigit():
    raise ValueError("Quantidade inválida ou vazia.")
quantidade = int(qtd_raw)

if quantidade <= 0:
    raise ValueError("Quantidade deve ser maior que zero.")

# Unidade de medida (sempre obrigatória)
unidade = str(row["unidade de medida"]).strip()
if not unidade:
    raise ValueError("Unidade de medida não pode ser vazia.")

# Descrição: se coluna existe e há valor não vazio, usa; senão, concatena nome + unidade
if "descrição" in row and pd.notna(row["descrição"]) and str(row["descrição"]).strip():
    descricao = str(row["descrição"]).strip()
else:
    descricao = f"{produto_raw} {unidade}"

# Marca (opcional)
marca = (
    str(row.get("marca", "")).strip() if pd.notna(row.get("marca")) else None
)

validade = self._parse_date(row.get("validade"), produto_raw)

# Categoria
cat_raw = str(row["categoria"]).strip()
normalized_cat = normalize_name(cat_raw)
categoria_id = self.category_map.get(normalized_cat)
if not categoria_id:
    # Em teoria, todas as categorias já foram criadas em _fetch_or_create_categories()
    raise ValueError(f"Falha ao encontrar ou criar a categoria '{cat_raw}'.")

# --- 2. Normalização e busca de duplicata ---
nome_normalizado = normalize_name(produto_raw)
existing = await ItemFinder.find_exact_match(
    self.db, nome_normalizado, validade, categoria_id, marca
)

# --- 3. Se duplicata existe, atualiza; senão, cria novo ---
if existing:
    existing.quantidade_item += quantidade
    existing.data_entrada_item = datetime.now()
    if validade:
        existing.data_validade_item = validade
    if marca:
        existing.marca_item = marca
    # Atualiza descrição também, caso queira manter histórico (opcional)
    existing.descricao_item = descricao
    existing.auditoria_usuario_id = self.auditoria_usuario_id
    await self.db.flush()
    self._items_updated += 1
else:
    from models.item import Item
    new_item = Item(
```

```
                nome_item_original=produto_raw,
                nome_item=nome_normalizado,
                descricao_item=descricao,
                unidade_medida_item=unidade,
                quantidade_item=quantidade,
                categoria_id=categoria_id,
                data_validade_item=validade,
                marca_item=marca,
                data_entrada_item=datetime.now(),
                auditoria_usuario_id=self.auditoria_usuario_id,
            )
            self.db.add(new_item)
            await self.db.flush()
            self._items_created += 1

    def _parse_date(self, raw_value, produto_raw: str):
        """Tenta converter a coluna 'validade' em date; levanta ValueError se inválido."""
        from datetime import datetime as dt

        if raw_value is pd.NA or pd.isna(raw_value):
            return None

        if isinstance(raw_value, (dt,)):
            return raw_value.date()

        # Tentar formato dd/mm/YYYY
        parsed = pd.to_datetime(raw_value, errors="coerce", format="%d/%m/%Y")
        if pd.isna(parsed):
            # Tentar inferir
            parsed = pd.to_datetime(raw_value, errors="coerce")
        if pd.isna(parsed):
            raise ValueError(f"Formato de validade inválido para '{produto_raw}': {raw_value}")
        return parsed.date()
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\categoria_service.py

```
# services/categoria_service.py
from schemas.categoria import CategoriaCreate, CategoriaUpdate, PaginatedCategorias, CategoriaOut
from repositories.categoria_repository import CategoriaRepository
from fastapi import HTTPException, status
from sqlalchemy.exc import IntegrityError
from sqlalchemy.ext.asyncio import AsyncSession
from utils.normalizar_texto import normalize_name
import math


class CategoriaService:

    @staticmethod
    async def create_categoria(db: AsyncSession, categoria_data: CategoriaCreate):
        try:
            # Normaliza o nome
            nome_original = categoria_data.nome_categoria.strip()
            nome_normalizado = normalize_name(nome_original)

            # Verifica se já existe categoria com o nome normalizado
            categoria_existente = await CategoriaRepository.aux_get_categoria_by_name(
                db, nome_normalizado
            )
            if categoria_existente:
```

```
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="Já existe uma categoria com este nome"
            )

        # Prepara os dados para criação
        dados_categoria = categoria_data.model_dump()
        dados_categoria.update({
            "nome_original": nome_original,
            "nome_categoria": nome_normalizado
        })

        return await CategoriaRepository.create_categoria(db, dados_categoria)

    except IntegrityError as e:
        await db.rollback()
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Erro de integridade ao criar categoria"
        )


@staticmethod
async def get_categorias(db: AsyncSession):
    categorias = await CategoriaRepository.get_categorias(db)
    return categorias

@staticmethod
async def get_categoria_by_id(db: AsyncSession, categoria_id: int):
    categoria = await CategoriaRepository.get_categoria_by_id(db, categoria_id)
    if not categoria:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Categoria não encontrada"
    return categoria

@staticmethod
async def get_categoria_by_name(db: AsyncSession, categotia_name: str):
    normalized_name = normalize_name(categotia_name)
    categoria = await CategoriaRepository.get_categoria_by_name(db, normalized_name)
    print(normalized_name)
    if not categoria:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Categoria não encontrada"
    return categoria

@staticmethod
async def get_categorias_like(db: AsyncSession, termo_busca: str):
    """
    Busca categorias cujos nomes contenham o termo de busca (case-insensitive)

    Args:
        db: Sessão async do SQLAlchemy
        termo_busca: String com o termo a ser buscado (ex: "papel")

    Returns:
        Lista de objetos Categoria que correspondem à busca
    """
    termo_normalizado = normalize_name(termo_busca)

    # Obtém o resultado da query (ainda não consumido)
    result = await CategoriaRepository.get_categoria_by_name_like(db, termo_normalizado)

    # Converte para lista de objetos Categoria
    categorias = result.scalars().all()
    if not categorias:
```

```
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Nenhuma categoria encontrada com o termo fornecido"
            )

        return categorias

    @staticmethod
    async def update_categoria(db: AsyncSession, categoria_id: int, update_data: CategoriaUpdate):
        # Busca a categoria existente
        categoria = await CategoriaRepository.get_categoria_by_id(db, categoria_id)

        # Converte para dicionário e remove campos não setados
        update_values = update_data.model_dump(exclude_unset=True)

        if 'nome_categoria' in update_values:
            # Processa novo nome
            novo_original = update_values['nome_categoria'].strip()
            novo_normalizado = normalize_name(novo_original)

            update_values['nome_original'] = novo_original
            update_values['nome_categoria'] = novo_normalizado

            # Remove o campo temporário
            del update_values['nome_categoria']

        # Atualiza apenas os campos permitidos
        return await CategoriaRepository.update_categoria(db, categoria_id, update_values)

    @staticmethod
    async def delete_categoria(db: AsyncSession, categoria_id: int):
        result = await CategoriaRepository.delete_categoria(db, categoria_id)
        if not result:
            raise HTTPException(status_code=404, detail="Categoria não encontrada")
        return result

    #função para retorno de categorias paginadas
    @staticmethod
    async def get_categorias_paginated(
        db: AsyncSession,
        page: int,
        size: int
    ) -> PaginatedCategorias:
        # validação de tamanho
        allowed = [5, 10, 25, 50, 100]
        if size not in allowed:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail=f"size deve ser um de {allowed}"
            )
        if page < 1:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="page deve ser >= 1"
            )

        # conta total de categorias
        total = await CategoriaRepository.count_categorias(db)
        offset = (page - 1) * size
        itens = await CategoriaRepository.get_categorias_paginated(db, offset, size)

        # converte para DTO
        items_out = [CategoriaOut.model_validate(i) for i in itens]
```

```
    # calcula total de páginas
    total_pages = math.ceil(total / size) if total > 0 else 1

    return PaginatedCategorias(
        page=page,
        size=size,
        total=total,
        total_pages=total_pages,
        items=items_out
    )

@staticmethod
async def search_categorias_paginated(
    db: AsyncSession,
    nome_categoria: str | None,
    page: int,
    size: int
) -> PaginatedCategorias:
    # validações (idênticas a get_categorias_paginated)
    allowed = [5, 10, 25, 50, 100]
    if size not in allowed:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=f"size deve ser um de {allowed}"
        )
    if page < 1:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="page deve ser >= 1"
        )

    # normaliza e traduz nome_categoria em IDs
    nome_norm = normalize_name(nome_categoria) if nome_categoria else None
    categoria_ids = None
    if nome_categoria:
        nome_cat_norm = normalize_name(nome_categoria)
        categoria_ids = await CategoriaRepository.find_categoria_ids_by_name(db, nome_cat_norm)

    # conta total de itens filtrados
    total = await CategoriaRepository.count_filtered_categorias(
        db, categoria_ids=categoria_ids, nome_categoria_normalizado=nome_norm
    )

    # calcula offset e traz só a página
    offset = (page - 1) * size
    categorias = await CategoriaRepository.get_filtered_categorias_paginated(
        db, categoria_ids=categoria_ids,
        nome_categorias_normalizado=nome_norm,
        offset=offset, limit=size
    )

    categorias_out = [CategoriaOut.model_validate(i) for i in categorias]
    total_pages = math.ceil(total / size) if total > 0 else 1

    return PaginatedCategorias(
        page=page, size=size,
        total=total, total_pages=total_pages,
        items=categorias_out
    )
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\export_strategy.py

```
# services\export_strategy.py

from abc import ABC, abstractmethod
import pandas as pd

class ExportStrategy(ABC):
    @abstractmethod
    def export(self, df: pd.DataFrame, file_path: str):
        pass

class CSVExportStrategy(ExportStrategy):
    def export(self, df: pd.DataFrame, file_path: str):
        # Para CSV, usar encoding='utf-8-sig' que adiciona BOM
        # Isso ajuda o Excel a reconhecer UTF-8 automaticamente ao abrir o CSV
        df.to_csv(file_path, index=False, sep=';', encoding='utf-8-sig')

class XLSXExportStrategy(ExportStrategy):
    def export(self, df: pd.DataFrame, file_path: str):
        with pd.ExcelWriter(file_path, engine='openpyxl') as writer:
            df.to_excel(writer, index=False, sheet_name='Relatorio Items')
            worksheet = writer.sheets['Relatorio Items']
            # col_widths = [15, 30, 40, 15, 20] # Remova ou ajuste esta linha se não for mais útil
            # Para XLSX, openpyxl já lida bem com UTF-8, mas ajustes de largura são bons.
            for col in worksheet.columns:
                max_length = 0
                column = [cell for cell in col] # Garante que 'column' é uma lista iterável de células
                try:
                    max_length = max(len(str(cell.value)) for cell in column)
                except ValueError: # Trata caso de coluna vazia ou com valores não-str
                    pass
                # Ajuste para uma largura mínima razoável
                adjusted_width = max(len(str(column[0].value)) if column and column[0].value else 10, ma
                worksheet.column_dimensions[column[0].column_letter].width = adjusted_width
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\finder.py

```
# services/item/finder.py

from datetime import date, datetime
from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession
from models.item import Item


class ItemFinder:
    @staticmethod
    async def find_exact_match(
        db: AsyncSession,
        nome_item_normalizado: str,
        validade: date | datetime | None,
        categoria_id: int,
        marca_item: str | None,
    ) -> Item | None:
        query = select(Item).where(
            Item.nome_item == nome_item_normalizado,
```

```
            Item.categoria_id == categoria_id,
        )
        if validade is not None:
            query = query.where(Item.data_validade_item == validade)
        else:
            query = query.where(Item.data_validade_item.is_(None))
        if marca_item is not None:
            query = query.where(Item.marca_item == marca_item)
        else:
            query = query.where(Item.marca_item.is_(None))

        result = await db.execute(query)
        return result.scalars().first()
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\item_service.py

```
# app/services/item/item_service.py

from datetime import datetime
from sqlalchemy.exc import IntegrityError
from sqlalchemy.ext.asyncio import AsyncSession
from fastapi import HTTPException, status, UploadFile
from utils.normalizar_texto import normalize_name
from services.validator import ItemValidator
from services.finder import ItemFinder
from services.bulk_processor import ItemBulkProcessor
from repositories.item_repository import ItemRepository
from repositories.categoria_repository import CategoriaRepository
from schemas.item import (
    ItemCreate,
    ItemUpdate,
    PaginatedItems,
    ItemOut,
    BulkItemUploadResult,
)

class ItemService:

    @staticmethod
    async def create_item(db: AsyncSession, item_data: ItemCreate, current_user):
        # 1) Valida campos obrigatórios
        ItemValidator.validate_on_create(item_data)

        # 2) Normaliza nome
        nome_original = item_data.nome_item.strip()
        nome_normalizado = normalize_name(nome_original)

        # 3) Prepara dicionário para persistência
        dados = item_data.model_dump()
        dados.update(
            {
                "nome_item_original": nome_original,
                "nome_item": nome_normalizado,
                "auditoria_usuario_id": current_user.usuario_id,
            }
        )
        # Se não vier data_entrada_item, usa agora
        dados["data_entrada_item"] = dados.get("data_entrada_item") or datetime.now()

        try:
```

```
            # 4) Verifica duplicata (incluindo itens inativos para reativação)
            existing = await ItemFinder.find_exact_match(
                db,
                nome_normalizado,
                dados.get("data_validade_item"),
                dados["categoria_id"],
                dados.get("marca_item"),
            )

            if existing:
                # Se existir duplicado, incrementa quantidade e reativa se necessário
                return await ItemService._increment_existing_item(db, existing, item_data)

            # 5) Se não, cria via repositório
            from models.item import Item # Importação local para evitar circular
            novo = Item(**dados)
            return await ItemRepository.create(db, novo)

        except IntegrityError as ie:
            await db.rollback()
            ItemService._handle_integrity_error(ie)
        except HTTPException:
            raise
        except Exception as e:
            raise HTTPException(
                status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
                detail=f"Erro ao criar o item: {e}",
            )

    @staticmethod
    async def _increment_existing_item(db: AsyncSession, existing, item_data: ItemCreate):
        """
        Incrementa apenas a quantidade (e atualiza campos opcionais)
        sem criar novo registro. Reativa o item se ele estava inativo.
        """
        existing.quantidade_item += item_data.quantidade_item
        existing.data_entrada_item = item_data.data_entrada_item or datetime.now()

        # Atualiza campos opcionais, se vierem
        if item_data.data_validade_item:
            existing.data_validade_item = item_data.data_validade_item
        if item_data.quantidade_minima_item:
            existing.quantidade_minima_item = item_data.quantidade_minima_item
        if item_data.marca_item:
            existing.marca_item = item_data.marca_item

        # Se o item estava inativo e sua quantidade foi incrementada, reativá-lo
        if not existing.ativo:
            existing.ativo = True

        existing.auditoria_usuario_id = (
            item_data.auditoria_usuario_id
            if hasattr(item_data, "auditoria_usuario_id")
            else existing.auditoria_usuario_id
        )

        await db.commit()
        await db.refresh(existing)
        return existing

    @staticmethod
    async def get_itens(db: AsyncSession):
        items = await ItemRepository.get_all(db)
```

```
    if not items:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Sem itens no banco de dados"
        )
    return items

@staticmethod
async def get_item_by_id(db: AsyncSession, item_id: int):
    item = await ItemRepository.get_by_id(db, item_id)
    if not item:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Item não encontrado"
        )
    return item

@staticmethod
async def delete_item(db: AsyncSession, item_id: int):
    item = await ItemRepository.get_by_id(db, item_id)
    if not item:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Item não encontrado"
        )
    # O ItemRepository.delete já realiza o soft delete (seta ativo=False)
    await ItemRepository.delete(db, item)
    return {"message": "Item deletado com sucesso"}

@staticmethod
async def update_item(
    db: AsyncSession, item_id: int, data: ItemUpdate, current_user
):
    # 1) Busca o item (get_by_id já filtra por ativo=True)
    item = await ItemRepository.get_by_id(db, item_id)
    if not item:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Item não encontrado"
        )

    # 2) Valida campos que vieram
    ItemValidator.validate_on_update(data)

    # 3) Se vier nome item, normaliza
    valores = data.model_dump(exclude_unset=True)
    if "nome_item" in valores:
        nome_original = valores["nome_item"].strip()
        valores["nome_item_original"] = nome_original
        valores["nome_item"] = normalize_name(nome_original)

    # 4) Verificar duplicata com novos valores (inclui inativos)
    novo_nome = valores.get("nome_item", item.nome_item)
    nova_marca = valores.get("marca_item", item.marca_item)
    nova_validade = valores.get("data_validade_item", item.data_validade_item)
    nova_categoria = valores.get("categoria_id", item.categoria_id)

    existing = await ItemFinder.find_exact_match(
        db, novo_nome, nova_validade, nova_categoria, nova_marca
    )

    # 5a) Merging: se encontrou uma duplicata diferente do item atual
    if existing and existing.item_id != item.item_id:
        # Transfere quantidade para o item existente (duplicata)
        existing.quantidade_item += item.quantidade_item
        # Reativa o item existente se ele estava inativo
        if not existing.ativo:
```

```
                existing.ativo = True

            # Atualiza apenas campos opcionais do item existente, se vierem
            for campo in ["quantidade_minima_item", "data_validade_item", "marca_item", "unidade_medida_
                if campo in valores:
                    setattr(existing, campo, valores[campo])

            existing.auditoria_usuario_id = current_user.usuario_id

            # O item original (item_id) é soft-deletado
            await ItemRepository.delete(db, item) # Já seta item.ativo = False
            await db.commit()
            await db.refresh(existing)
            return existing
        # 5b) Senão, faz atualização pontual no próprio item
        for key, valor in valores.items():
            setattr(item, key, valor)

        # Lógica para reativar o item se a quantidade for > 0 ou se 'ativo' for explicitamente True
        if 'quantidade_item' in valores and valores['quantidade_item'] > 0 and not item.ativo:
            item.ativo = True
        elif 'ativo' in valores: # Permite definir o status ativo/inativo explicitamente
            item.ativo = valores['ativo']

        item.auditoria_usuario_id = current_user.usuario_id

        await db.commit()
        await db.refresh(item)
        return item

    @staticmethod
    async def get_items_paginated(db: AsyncSession, page: int, size: int) -> PaginatedItems:
        allowed = [5, 10, 25, 50, 100]
        if size not in allowed:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail=f"size deve ser um de {allowed}",
            )
        if page < 1:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST, detail="page deve ser >= 1"
            )

        total = await ItemRepository.count(db) # Já conta apenas itens ativos
        offset = (page - 1) * size
        itens = await ItemRepository.get_paginated(db, offset, size) # Já lista apenas itens ativos

        items_out = [ItemOut.model_validate(i) for i in itens]
        total_pages = (total // size) + (1 if total % size else 0)

        return PaginatedItems(
            page=page, size=size, total=total, total_pages=total_pages, items=items_out
        )

    @staticmethod
    async def search_items_paginated(
        db: AsyncSession, nome_produto: str | None, nome_categoria: str | None, page: int, size: int
    ) -> PaginatedItems:
        allowed = [5, 10, 25, 50, 100]
        if size not in allowed:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail=f"size deve ser um de {allowed}",
```

```
            )
        if page < 1:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST, detail="page deve ser >= 1"
            )

        nome_norm = normalize_name(nome_produto) if nome_produto else None
        categoria_ids = None
        if nome_categoria:
            nome_categoria_norm = normalize_name(nome_categoria)
            categoria_ids = await CategoriaRepository.find_categoria_ids_by_name(
                db, nome_categoria_norm
            )

        total = await ItemRepository.count_filtered(db, categoria_ids, nome_norm) # Já conta apenas iten
        offset = (page - 1) * size
        itens = await ItemRepository.get_filtered_paginated( # Já lista apenas itens ativos
            db, categoria_ids, nome_norm, offset, size
        )

        items_out = [ItemOut.model_validate(i) for i in itens]
        total_pages = (total // size) + (1 if total % size else 0)

        return PaginatedItems(
            page=page, size=size, total=total, total_pages=total_pages, items=items_out
        )

    @staticmethod
    async def process_bulk_upload(
        db: AsyncSession, file: UploadFile, auditoria_usuario_id: int
    ) -> BulkItemUploadResult:
        processor = ItemBulkProcessor(db, auditoria_usuario_id)
        return await processor.process(file)

    @staticmethod
    def _handle_integrity_error(e: IntegrityError):
        error_msg = str(e.orig).lower()
        if "fk_item_categoria" in error_msg:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="Categoria do item não encontrada no banco de dados.",
            )
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Erro ao criar/atualizar o item. Verifique os dados e tente novamente."
        )
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\relatorio_service.py

```
#services\relatorio_service.py

from sqlalchemy.ext.asyncio import AsyncSession
import pandas as pd
from models.retirada import StatusEnum
from core.configs import Settings
from services.export_strategy import CSVExportStrategy, XLSXExportStrategy
from services.categoria_service import CategoriaService
from repositories.item_repository import ItemRepository
from services.retirada_service import RetiradaService
from utils.relatorio_itens import formatar_dados_relatorio
```

```
from fastapi import HTTPException
from datetime import datetime
import os
import unicodedata
from utils.normalizar_texto import normalize_name

PASTA_RELATORIOS = Settings.PASTA_RELATORIOS
os.makedirs(PASTA_RELATORIOS, exist_ok=True)

class RelatorioService:

    @staticmethod
    async def gerar_relatorio_quantidade_itens(
        session: AsyncSession,
        filtro_categoria: str = None,
        filtro_produto: str = None,
        formato: str = "csv"
    ):
        try:
            # 1. Tratar filtro de categoria: ID numérico ou texto
            categoria_ids = []
            if filtro_categoria:
                if filtro_categoria.isdigit():
                    cat = await CategoriaService.get_categoria_by_id(session, int(filtro_categoria))
                    if not cat:
                        raise HTTPException(status_code=404, detail="Categoria não encontrada")
                    categoria_ids = [cat.categoria_id]
                else:
                    cats = await CategoriaService.get_categorias_like(session, filtro_categoria)
                    if not cats:
                        raise HTTPException(status_code=404, detail="Nenhuma categoria encontrada com o
                    categoria_ids = [c.categoria_id for c in cats]

            # 2. Normalizar e tratar filtro de produto
            filtro_normalizado = None
            if filtro_produto:
                filtro_normalizado = normalize_name(filtro_produto)

            # 3. Buscar itens
            itens = await ItemRepository.find_filtered(
                session,
                categoria_ids=categoria_ids or None,
                nome_produto_normalizado=filtro_normalizado
            )

            # 4. Formatar e exportar
            dados = formatar_dados_relatorio(itens)
            df = pd.DataFrame(dados)

            caminho_arquivo = os.path.join(
                PASTA_RELATORIOS,
                f"relatorio_quantidade_itens.{formato}"
            )
            export_strategy = CSVExportStrategy() if formato == "csv" else XLSXExportStrategy()
            export_strategy.export(df, caminho_arquivo)

            return caminho_arquivo

        except HTTPException:
            raise
        except Exception as e:
            raise HTTPException(status_code=500, detail=f"Erro ao gerar relatório: {e}")
    @staticmethod
```

```
async def gerar_relatorio_entrada_itens(
    session: AsyncSession,
    data_inicio: datetime,
    data_fim: datetime,
    formato: str
):
    itens = await ItemRepository.get_items_period(session, data_inicio, data_fim) # Usando ItemRepos

    # Formatar DataFrame acessando os dicionários retornados pelo repositório
    df = pd.DataFrame([{
        "ID_Item": item["item_id"],
        "Nome": item["nome_item_original"],
        "Quantidade": item["quantidade_item"],
        "Data_Entrada": item["data_entrada_item"].strftime('%d/%m/%Y'),
        "Categoria": item["nome_categoria_original"] # Agora acessa a chave correta do dicionário
    } for item in itens])

    # Exportar usando estratégia existente
    caminho_arquivo = os.path.join(Settings.PASTA_RELATORIOS, f"relatorio_entrada_itens.{formato}")
    export_strategy = CSVExportStrategy() if formato == "csv" else XLSXExportStrategy()
    export_strategy.export(df, caminho_arquivo)

    return caminho_arquivo

@staticmethod
async def gerar_relatorio_retiradas_setor(
    session: AsyncSession,
    setor_id: int,
    data_inicio: datetime,
    data_fim: datetime,
    formato: str
):
    if not setor_id:
        raise HTTPException(status_code=400, detail="Setor não informado")

    retiradas = await RetiradaService.get_retiradas_por_setor_periodo(
        session, setor_id, data_inicio, data_fim
    )

    dados = []
    for retirada in retiradas:
        if not retirada.itens:
            continue
        for item in retirada.itens:
            if not item.item:
                continue
            dados.append({
                "ID_Retirada": retirada.retirada_id,
                "Data_Solicitacao": retirada.data_solicitacao.strftime('%d/%m/%Y'),
                "Item": item.item.nome_item,
                "Quantidade_Retirada": item.quantidade_retirada,
                "Usuario": retirada.usuario.nome_usuario,
                "Status": StatusEnum(retirada.status).name,
                "Autorizada_Por": retirada.admin.nome_usuario if retirada.admin else "N/A",
                "Setor_ID": retirada.setor_id
            })

    df = pd.DataFrame(dados)
    caminho_arquivo = os.path.join(PASTA_RELATORIOS, f"relatorio_retiradas_setor.{formato}")
    export_strategy = CSVExportStrategy() if formato == "csv" else XLSXExportStrategy()
    export_strategy.export(df, caminho_arquivo)

    return caminho_arquivo
```

```
    @staticmethod
    async def gerar_relatorio_retiradas_usuario(
        session: AsyncSession,
        usuario_id: int,
        data_inicio: datetime,
        data_fim: datetime,
        formato: str
    ):
        try:
            retiradas = await RetiradaService.get_retiradas_por_usuario_periodo(
                session, usuario_id, data_inicio, data_fim
            )

            dados = []
            for retirada in retiradas:
                for item in retirada.itens:
                    dados.append({
                        "ID_Retirada": retirada.retirada_id,
                        "Data_Solicitacao": retirada.data_solicitacao.strftime('%d/%m/%Y'),
                        "Item": item.item.nome_item,
                        "Marca": item.item.marca_item,
                        "Quantidade_Retirada": item.quantidade_retirada,
                        "Usuario_Retirou_ID": retirada.usuario.usuario_id,
                        "Usuario_Retirou_Nome": retirada.usuario.nome_usuario,
                        "Usuario_Retirou_SIAPE": retirada.usuario.siape_usuario or "N/A",
                        "Usuario_Autorizou_ID": retirada.admin.usuario_id if retirada.admin else None,
                        "Usuario_Autorizou_Nome": retirada.admin.nome_usuario if retirada.admin else "N/
                        "Usuario_Autorizou_SIAPE": retirada.admin.siape_usuario if retirada.admin else "
                        "Status": StatusEnum(retirada.status).name
                    })

            df = pd.DataFrame(dados)
            caminho_arquivo = os.path.join(
                PASTA_RELATORIOS,
                f'relatorio_retiradas_usuario_{usuario_id}_{datetime.now().timestamp()}.{formato}'
            )
            export_strategy = CSVExportStrategy() if formato == "csv" else XLSXExportStrategy()
            export_strategy.export(df, caminho_arquivo)

            return caminho_arquivo

        except Exception as e:
            raise HTTPException(
                status_code=500,
                detail=f"Erro ao gerar relatório: {str(e)}"
            )
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\retirada_service.py

```
#services/retirada_service.py

from fastapi import HTTPException, status
from sqlalchemy.ext.asyncio import AsyncSession
from datetime import datetime

from schemas.retirada import RetiradaOut
from models.retirada import Retirada, StatusEnum
from models.retirada_item import RetiradaItem
from repositories.retirada_repository import RetiradaRepository
from schemas.retirada import RetiradaCreate, RetiradaUpdateStatus, RetiradaPaginated, RetiradaFilterPara
```

```
from services.alerta_service import AlertaService
from utils.websocket_endpoints import manager

from models.usuario import Usuario, RoleEnum
from sqlalchemy.future import select

class RetiradaService:

    @staticmethod
    async def get_retiradas_paginadas(
        db: AsyncSession, page: int, page_size: int
    ) -> RetiradaPaginated:
        """Retorna uma lista paginada de todas as retiradas ativas."""
        total = await RetiradaRepository.count_retiradas(db)
        pages = (total + page_size - 1) // page_size if total > 0 else 1
        offset = (page - 1) * page_size
        sqlalchemy_items = await RetiradaRepository.get_retiradas_paginated(db, offset, page_size)
        items = [RetiradaOut.model_validate(ent) for ent in sqlalchemy_items]
        return RetiradaPaginated(total=total, page=page, pages=pages, items=items)

    @staticmethod
    async def filter_retiradas_paginated(
        db: AsyncSession,
        params: RetiradaFilterParams,
        page: int,
        page_size: int
    ) -> RetiradaPaginated:
        """Filtra e retorna retiradas ativas com paginação."""
        total = await RetiradaRepository.count_retiradas_filter(db, params)
        pages = (total + page_size - 1) // page_size if total > 0 else 1
        offset = (page - 1) * page_size
        sqlalchemy_items = await RetiradaRepository.filter_retiradas_paginated(
            db, params, offset, page_size
        )
        items = [RetiradaOut.model_validate(ent) for ent in sqlalchemy_items]
        return RetiradaPaginated(total=total, page=page, pages=pages, items=items)

    @staticmethod
    async def solicitar_retirada(db: AsyncSession, retirada_data: RetiradaCreate, usuario_id: int):
        """Cria uma nova solicitação de retirada e seus itens associados."""
        try:
            #1) Cria Retirada
            nova_retirada = Retirada(
                usuario_id=usuario_id,
                setor_id=retirada_data.setor_id,
                status=StatusEnum.PENDENTE,
                solicitado_localmente_por=retirada_data.solicitado_localmente_por,
                justificativa=retirada_data.justificativa,
                is_active=True # Garante que a nova retirada é criada como ativa
            )
            await RetiradaRepository.criar_retirada(db, nova_retirada)

            #2) Cria e adiciona itens
            itens_retirada = [
                RetiradaItem(
                    retirada_id=nova_retirada.retirada_id,
                    item_id=item.item_id,
                    quantidade_retirada=item.quantidade_retirada
                )
                for item in retirada_data.itens
            ]
            await RetiradaRepository.adicionar_itens_retirada(db, itens_retirada)
            #3) Commit
```

```
            await db.commit()

            #4) Recarrega com eager-load para serialização segura
            retirada_completa = await RetiradaRepository.buscar_retirada_por_id(
                db, nova_retirada.retirada_id
            )

            # Transmitir o evento de nova solicitação de retirada APENAS para usuários do Almoxarifado
            # Consulta para obter IDs de todos os usuários com o perfil de Almoxarifado
            almoxarifados = await db.execute(
                select(Usuario).where(Usuario.tipo_usuario == RoleEnum.USUARIO_ALMOXARIFADO.value)
            )
            for almoxarifado_user in almoxarifados.scalars().all():
                await manager.send_to_user(
                    almoxarifado_user.usuario_id,
                    {
                        "type": "new_withdrawal_request",
                        "retirada_id": retirada_completa.retirada_id,
                        "message": f"Nova solicitação de retirada do setor {retirada_completa.setor_id}"
                    }
                )

            return retirada_completa
        except Exception as e:
            await db.rollback()
            raise HTTPException(
                status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
                detail=f"Erro ao solicitar retirada: {e}"
            )

    @staticmethod
    async def atualizar_status(db: AsyncSession, retirada_id: int, status_data: RetiradaUpdateStatus, ad
        """Atualiza o status de uma retirada e, se concluída, decrementa o estoque dos itens."""
        try:
            if status_data.status not in (s.value for s in StatusEnum):
                raise HTTPException(400, "Status inválido.")

            retirada = await RetiradaRepository.buscar_retirada_por_id(db, retirada_id)

            if not retirada:
                raise HTTPException(status.HTTP_404_NOT_FOUND, "Retirada não encontrada")

            # Se concluindo, decrementa estoques
            if status_data.status == StatusEnum.CONCLUIDA:
                for ri in retirada.itens:
                    item = await RetiradaRepository.buscar_item_por_id(db, ri.item_id)
                    if not item: # Garante que o item existe antes de tentar acessar seus atributos
                        raise HTTPException(
                            status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"Item com ID {ri.item_id} não encontrado."
                        )
                    if item.quantidade_item < ri.quantidade_retirada:
                        raise HTTPException(
                            status_code=status.HTTP_400_BAD_REQUEST,
                            detail=f"Estoque insuficiente para item {item.nome_item_original}. Quantidad
                        )
                    await RetiradaRepository.atualizar_quantidade_item(
                        db, item, item.quantidade_item - ri.quantidade_retirada
                    )
                    # Se a quantidade do item chegar a 0, marcar como inativo (soft delete)
                    if item.quantidade_item == 0:
                        item.ativo = False # Marca o item como inativo (soft delete para item)
                        await db.flush() # Garante que a mudança seja persistida antes do commit
```

```
                await AlertaService.verificar_estoque_baixo(db, ri.item_id)

        retirada.status = status_data.status
        retirada.detalhe_status = status_data.detalhe_status
        retirada.autorizado_por = admin_id # Define quem autorizou/negou

        updated_retirada = await RetiradaRepository.atualizar_retirada(db, retirada)

        # Enviar notificação específica para o usuário que solicitou a retirada
        await manager.send_to_user(
            updated_retirada.usuario_id,
            {
                "type": "withdrawal_status_update",
                "retirada_id": updated_retirada.retirada_id,
                "status": updated_retirada.status,
                "message": f"Sua solicitação de retirada ID {updated_retirada.retirada_id} foi atual
            }
        )

        return updated_retirada
    except HTTPException:
        await db.rollback()
        raise
    except Exception as e:
        await db.rollback()
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail=f"Erro ao atualizar status: {e}"
        )

@staticmethod
async def get_retiradas_pendentes_paginated(
    db: AsyncSession, page: int, page_size: int
) -> RetiradaPaginated:
    """Retorna uma lista paginada de retiradas pendentes ativas."""
    total = await RetiradaRepository.count_retiradas_pendentes(db)
    pages = (total + page_size - 1) // page_size if total > 0 else 1
    offset = (page - 1) * page_size
    sqlalchemy_items = await RetiradaRepository.get_retiradas_pendentes_paginated(db, offset, page_s
    items = [RetiradaOut.model_validate(ent) for ent in sqlalchemy_items]
    return RetiradaPaginated(total=total, page=page, pages=pages, items=items)

@staticmethod
async def get_all_retiradas(db: AsyncSession):
    """Retorna todas as retiradas ativas."""
    try:
        all_r = await RetiradaRepository.get_retiradas(db)
        if not all_r:
            raise HTTPException(status.HTTP_404_NOT_FOUND, "Não há retiradas")
        return all_r
    except Exception as e:
        raise HTTPException(status.HTTP_500_INTERNAL_SERVER_ERROR, f"Erro: {e}")

@staticmethod
async def get_retirada_by_id(db: AsyncSession, retirada_id: int):
    """Busca uma retirada ativa pelo ID."""
    r = await RetiradaRepository.buscar_retirada_por_id(db, retirada_id)
    if not r:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Retirada não encontrada")
    return r

@staticmethod
async def get_retiradas_por_setor_periodo(db: AsyncSession, setor_id: int, data_inicio: datetime, da
```

```
        """Retorna retiradas ativas filtradas por setor e período."""
        try:
            res = await RetiradaRepository.get_retiradas_por_setor_periodo(db, setor_id, data_inicio, da
            if not res:
                raise HTTPException(status.HTTP_404_NOT_FOUND, "Nenhuma retirada nesse período/setor")
            return res
        except Exception as e:
            raise HTTPException(status.HTTP_500_INTERNAL_SERVER_ERROR, f"Erro: {e}")

    @staticmethod
    async def get_retiradas_por_usuario_periodo(db: AsyncSession, usuario_id: int, data_inicio: datetime
        """Retorna retiradas ativas filtradas por usuário e período."""
        try:
            res = await RetiradaRepository.get_retiradas_por_usuario_periodo(db, usuario_id, data_inicio
            if not res:
                raise HTTPException(status.HTTP_404_NOT_FOUND, "Nenhuma retirada para esse usuário")
            return res
        except Exception as e:
            raise HTTPException(status.HTTP_500_INTERNAL_SERVER_ERROR, f"Erro: {e}")

    @staticmethod
    async def get_retiradas_by_user_paginated(
        db: AsyncSession, usuario_id: int, page: int, page_size: int
    ) -> RetiradaPaginated:
        """Retorna uma lista paginada de retiradas ativas para um usuário específico."""
        total = await RetiradaRepository.count_retiradas_by_user(db, usuario_id)
        pages = (total + page_size - 1) // page_size if total > 0 else 1
        offset = (page - 1) * page_size
        sqlalchemy_items = await RetiradaRepository.get_retiradas_by_user_paginated(db, usuario_id, offs
        items = [RetiradaOut.model_validate(ent) for ent in sqlalchemy_items]
        return RetiradaPaginated(total=total, page=page, pages=pages, items=items)

    @staticmethod
    async def soft_delete_retiradas_by_period(db: AsyncSession, start_date: datetime, end_date: datetime
        """
        Realiza o soft delete (inativa) retiradas dentro de um período específico.
        """
        # Adicionar validação de datas, se necessário (ex: data_inicio < data_fim)
        if start_date >= end_date:
            raise HTTPException(status.HTTP_400_BAD_REQUEST, "A data inicial deve ser anterior à data fi

        updated_count = await RetiradaRepository.soft_delete_by_period(db, start_date, end_date)

        if updated_count == 0:
            return {"message": f"Nenhuma retirada encontrada no período de {start_date.strftime('%d/%m/%

        return {"message": f"{updated_count} retiradas foram deletadas (inativadas) com sucesso no perío
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\setor_service.py

```
# services/setor_service.py
from sqlalchemy.orm import Session
from sqlalchemy.future import select
from schemas.setor import SetorCreate, SetorUpdate
from repositories.setor_repository import SetorRepository
from models.setor import Setor
from fastapi import HTTPException, status

class SetorService:
    """Adicionar lógica para só permitir deletar um setor se não tiver nenhum usuário associado a ele"""
```

```
    @staticmethod
    async def create_setor(db: Session, setor_data: SetorCreate):

        result = await SetorRepository.create_setor(db, setor_data)
        return result

    @staticmethod
    async def get_setores(db: Session):
        result = await SetorRepository.get_setores(db)
        return result

    @staticmethod
    async def get_setor_by_id(db: Session, setor_id: int):
        result = await SetorRepository.get_setor_by_id(db, setor_id)
        return result

    @staticmethod
    async def update_setor(db: Session, setor_id: int, setor_data: SetorUpdate):
        result = await SetorRepository.update_setor(db, setor_id, setor_data)
        return result

    @staticmethod
    async def delete_setor(db: Session, setor_id: int):
        result = await SetorRepository.delete_setor(db, setor_id)
        return result

    @staticmethod
    async def create_root_setor(db:Session):
        existing_setor = await db.execute(select(Setor))
        if existing_setor.scalars().first() is not None:
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail="O sistema já possui setores cadastrados"
            )

        setor_root = SetorCreate(
            nome_setor="Setor Root",
            descricao_setor="Setor criado apenas para início do sistema"
        )
        return await SetorRepository.create_setor(db, setor_root)
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\usuario_service.py

```
#services/usuario_service.py

from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.future import select

from models.usuario import Usuario
from models.setor import Setor
from schemas.usuario import UsuarioCreate, UsuarioUpdate
from repositories.usuario_repository import UsuarioRepository
from fastapi import HTTPException, status, Depends
from core.security import get_password_hash, verify_password, create_access_token
from fastapi.security import OAuth2PasswordRequestForm
from core.database import get_session
from models.usuario import RoleEnum
from services.setor_service import SetorService

class UsuarioService:
```

```
@staticmethod
async def create_first_user(db: AsyncSession, user_data: UsuarioCreate):
    # Verifica se já existe algum usuário no sistema
    existing_user = await db.execute(select(Usuario))
    if existing_user.scalars().first() is not None:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="O sistema já possui usuários cadastrados"
        )

    setor_root = await SetorService.create_root_setor(db)
    setor_root_result = await db.execute(select(Setor).where(Setor.setor_id == setor_root.setor_id))
    setor_root_data = setor_root_result.scalars().first()

    user_data.tipo_usuario = RoleEnum.USUARIO_DIRECAO.value
    user_data.setor_id = setor_root_data.setor_id

    user_root = await UsuarioService.create_usuario(db, user_data)
    return user_root

@staticmethod
async def create_usuario (db: AsyncSession, user_data: UsuarioCreate):
    """Cria um novo usuário após validar os dados."""
    await UsuarioService._validate_user_data(db, user_data)
    return await UsuarioRepository.create_usuario(db, user_data)

@staticmethod
async def get_usuarios (db: AsyncSession):
    """Retorna todos os usuários cadastrados"""
    return await UsuarioRepository.get_usuarios(db)

@staticmethod
async def get_usuario_by_id(db: AsyncSession, usuario_id: int):
    """Obtém um usuário pelo ID"""
    usuario = await UsuarioRepository.get_usuario_by_id(db, usuario_id)
    if not usuario:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Usuário não encontrado"
        )
    return usuario

@staticmethod
async def delete_usuario(db: AsyncSession, usuario_id: int, current_user: Usuario):
    usuario = await UsuarioRepository.get_usuario_by_id(db, usuario_id)
    if not usuario:
        raise HTTPException(
            detail="Usuário não encontrado",
            status_code=status.HTTP_404_NOT_FOUND,
        )
    # Lógica de permissão permanece aqui, no serviço
    if current_user.tipo_usuario != RoleEnum.USUARIO_DIRECAO.value:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Sem permissão para esta operação"
        )

    return await UsuarioRepository.delete_usuario(db, usuario_id)

@staticmethod
async def update_usuario(
    db: AsyncSession,
    usuario_id: int,
```

```
    usuario_data: UsuarioUpdate,
    current_user: Usuario
):
    """Atualiza os dados de um usuário"""
    usuario = await UsuarioRepository.get_usuario_by_id(db, usuario_id)
    if not usuario:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Usuário não encontrado")

    UsuarioService._validate_permission(usuario_id, current_user)
    await UsuarioService._validate_user_data(db, usuario_data, usuario_id)

    campos_atualizados = UsuarioService._prepare_update_fields(db, usuario, usuario_data)

    if campos_atualizados:
        await db.commit()
        await db.refresh(usuario)
    return usuario

@staticmethod
async def login_user(
    form_data: OAuth2PasswordRequestForm = Depends(),
    db: AsyncSession = Depends(get_session)
):
    """Realiza o login do usuário e retorna um token JWT"""
    user = await db.scalar(
        select(Usuario).where(Usuario.username == form_data.username)
    )
    if not user or not verify_password(form_data.password, user.senha_usuario):
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Credenciais inválidas"
        )

    access_token = create_access_token(
        data_payload={"sub": user.username},
        tipo_usuario=user.tipo_usuario,
        usuario_id=user.usuario_id
    )
    return {
        "access_token": access_token,
        "token_type": "bearer",
        "tipo_usuario": user.tipo_usuario,
        "usuario_id": user.usuario_id
    }

 # MÉTODO PARA REDEFINIÇÃO DE SENHA SIMPLES
@staticmethod
async def reset_password_simple(db: AsyncSession, username_or_email: str, new_password: str):
    """
    Busca um usuário por username ou email e redefine sua senha.
    LEMBRETE: Esta lógica é insegura para produção.
    """
    # Tenta encontrar por username
    user = await db.scalar(
        select(Usuario).where(Usuario.username == username_or_email)
    )

    # Se não encontrou por username, tenta por email
    if not user:
        user = await db.scalar(
            select(Usuario).where(Usuario.email_usuario == username_or_email.lower())
        )
    if not user:
```

```
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Usuário não encontrado."
            )

        # Atualiza a senha
        user.senha_usuario = get_password_hash(new_password)
        await db.commit()
        await db.refresh(user)

        return user

    # MÉTODO PARA CHECAR SE O USUÁRIO EXISTE PARA REDEFINIÇÃO DE SENHA
    @staticmethod
    async def check_user_exists_for_reset(db: AsyncSession, username_or_email: str) -> bool:
        """
        Verifica se um usuário existe com o username ou email fornecido.
        Retorna True se existe, False caso contrário.
        """
        # Tenta encontrar por username
        user = await db.scalar(
            select(Usuario).where(Usuario.username == username_or_email)
        )

        # Se não encontrou por username, tenta por email
        if not user:
            user = await db.scalar(
                select(Usuario).where(Usuario.email_usuario == username_or_email.lower())
            )

        return user is not None


    #----------- MÉTODOS AUXILIXARES ABAIXO ---------------------------

    @staticmethod
    async def _validate_user_data(
        db: AsyncSession,
        user_data: UsuarioCreate | UsuarioUpdate,
        exclude_usuario_id: int | None = None
    ):
        """Valida os dados do usuário (username, email, siape e tipo_usuario)"""
        await UsuarioService._validate_unique_fields(db, user_data, exclude_usuario_id)
        UsuarioService._validate_tipo_usuario(user_data.tipo_usuario)
        if isinstance(user_data, UsuarioCreate):
            await UsuarioService._validate_setor(db, user_data.setor_id)

    @staticmethod
    async def _validate_unique_fields(db: AsyncSession, usuario_data: UsuarioUpdate, exclude_usuario_id:
        """Valida se email, username e siape já estão em uso."""
        campos = {
            "username": usuario_data.username,
            "email_usuario": usuario_data.email_usuario,
            "siape_usuario": usuario_data.siape_usuario
        }
        for campo, valor in campos.items():
            if valor is not None and valor != '':
                query = await db.scalar(
                    select(Usuario).where(
                        getattr(Usuario, campo) == valor,
                        Usuario.usuario_id != exclude_usuario_id
                    )
                )
```

```
            if query:
                raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail=f"{campo.title()}

    @staticmethod
    def _validate_tipo_usuario(tipo_usuario: int):
        """Valida se o tipo de usuário está dentro dos valores permitidos."""
        if tipo_usuario not in [1, 2, 3]:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="Tipo de usuário não permitido"
            )

    @staticmethod
    async def _validate_setor (db: AsyncSession, setor_id: int):
        """ Verifica se o setor informado existe no banco de dados."""
        if not await db.get(Setor, setor_id):
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Setor não encontrado"
            )

    @staticmethod
    def _validate_permission (usuario_id: int, current_user: Usuario):
        """ Verifica se o usuário tem permissão para atualizar os dados. """
        # Direção (tipo 3) pode editar qualquer usuário.
        # Outros usuários só podem editar a si mesmos

        if current_user.tipo_usuario != RoleEnum.USUARIO_DIRECAO.value and usuario_id != current_user.us
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail="Sem permissão para esta operação"
            )

    @staticmethod
    def _prepare_update_fields (db: AsyncSession, usuario: Usuario, usuario_data: UsuarioUpdate):
        """Atualiza apenas os campos modificados."""
        campos_atualizados = False

        if usuario_data.nome_usuario is not None:
            usuario.nome_usuario = usuario_data.nome_usuario
            campos_atualizados = True

        if usuario_data.email_usuario is not None:
            usuario.email_usuario = usuario_data.email_usuario.lower()
            campos_atualizados = True

        if usuario_data.username is not None:
            usuario.username = usuario_data.username
            campos_atualizados = True

        if usuario_data.tipo_usuario is not None:
            usuario.tipo_usuario = usuario_data.tipo_usuario
            campos_atualizados = True

        if usuario_data.setor_id is not None:
            usuario.setor_id = usuario_data.setor_id
            campos_atualizados = True

        if usuario_data.siape_usuario is not None:
            usuario.siape_usuario = usuario_data.siape_usuario
            campos_atualizados = True
        elif hasattr(usuario_data, 'siape_usuario') and usuario_data.siape_usuario is None:
            usuario.siape_usuario = None
```

```
            campos_atualizados = True

        if usuario_data.senha_usuario:
            usuario.senha_usuario = get_password_hash(usuario_data.senha_usuario)
            campos_atualizados = True

        return campos_atualizados
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services\validator.py

```
# services/item/validator.py

from fastapi import HTTPException, status
from schemas.item import ItemCreate, ItemUpdate


class ItemValidator:
    @staticmethod
    def validate_on_create(item_data: ItemCreate) -> None:
        missing_fields = []

        if not item_data.nome_item or not item_data.nome_item.strip():
            missing_fields.append("nome_item")
        if not item_data.descricao_item or not item_data.descricao_item.strip():
            missing_fields.append("descricao_item")
        if not item_data.unidade_medida_item or not item_data.unidade_medida_item.strip():
            missing_fields.append("unidade_medida_item")
        if item_data.quantidade_item is None or item_data.quantidade_item <= 0:
            missing_fields.append("quantidade_item (deve ser maior que zero)")
        if item_data.categoria_id is None:
            missing_fields.append("categoria_id")

        if missing_fields:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail=f"Campos obrigatórios ausentes ou inválidos: {', '.join(missing_fields)}",
            )

    @staticmethod
    def validate_on_update(item_data: ItemUpdate) -> None:
        # Em update, só validamos se vierem campos específicos
        if item_data.quantidade_item is not None and item_data.quantidade_item <= 0:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="quantidade_item (deve ser maior que zero)",
            )

        # Se vier nome_item, verifica não vir vazio
        if item_data.nome_item is not None and not item_data.nome_item.strip():
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="nome_item não pode ser vazio",
            )
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\services__init__.py

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\utils\date_parser.py

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\utils\logger.py

```
# utils/logger.py
import logging
from logging.handlers import TimedRotatingFileHandler
from pathlib import Path
from core.configs import settings

# Diretório de logs: <projeto_root>/logs
LOG_DIR = settings.PROJECT_ROOT / "logs"
LOG_DIR.mkdir(exist_ok=True)

# Handler que gira o arquivo todo dia à meia■noite e adiciona sufixo YYYY■MM■DD.txt
handler = TimedRotatingFileHandler(
    filename=LOG_DIR / "app.log",
    when="midnight",
    interval=1,
    encoding="utf-8",
    backupCount=30,             # opcional: manter 30 dias
    utc=False
)
handler.suffix = "%Y-%m-%d.txt"

# Formato de log
formatter = logging.Formatter(
    "%(asctime)s %(levelname)s [%(name)s] %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S"
)
handler.setFormatter(formatter)

# Logger da aplicação
logger = logging.getLogger("almoxarifado")
logger.setLevel(logging.INFO)    # ou DEBUG
logger.addHandler(handler)

# propagar logs do Uvicorn/FastAPI:
for uv in ("uvicorn", "uvicorn.error", "uvicorn.access"):
    uvlog = logging.getLogger(uv)
    uvlog.handlers.clear()
    uvlog.addHandler(handler)
    uvlog.setLevel(logging.INFO)
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\utils\normalizar_texto.py

```
import unicodedata
import re

def normalize_name(name: str) -> str:
```

```
    """Normaliza o nome do item removendo acentos (exceto 'ç'), caracteres especiais e espaços."""
    if not name:
        return ""

    # Mantém apenas caracteres alfanuméricos e 'ç', removendo acentos de outras letras
    name = unicodedata.normalize('NFKD', name)  # Separa letras de acentos
    name = ''.join(char for char in name if char.isalnum() or char in "çÇ")  # Remove caracteres especia

    # Converte para uppercase e remove espaços
    return name.upper()
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\utils\relatorio_itens.py

```
#utils\relatorio_itens.py

import os
import pandas as pd
from sqlalchemy.ext.asyncio import AsyncSession
from fastapi import HTTPException
from core.configs import Settings
from services.categoria_service import CategoriaService
from services.export_strategy import CSVExportStrategy, XLSXExportStrategy
from services.item_service import ItemService



def get_pasta_relatorios() -> str:
    """Retorna o caminho absoluto para a pasta de relatórios"""
    return str(Settings.PASTA_RELATORIOS)

def gerar_dataframe_items(dados):
    """
    Cria um DataFrame a partir dos dados obtidos na consulta.
    """
    df = pd.DataFrame(dados)

    if not df.empty:
        # Formatar os nomes dos produtos e marcas
        df['Produto'] = (
            df['Produto']
            .str.title()
        )
        if 'Marca' in df.columns:
            df['Marca'] = df['Marca'].str.title()

        # Formatar data (se existir)
        if 'Data_Validade' in df.columns:
            df['Data_Validade'] = pd.to_datetime(df['Data_Validade']).dt.strftime('%d/%m/%Y')

    return df

def salvar_relatorio(df, formato):
    arquivo_nome = f"relatorio_quantidade_itens.{formato}"
    caminho_arquivo = os.path.join(get_pasta_relatorios(), arquivo_nome)

    try:
        if formato == "csv":
            df.to_csv(caminho_arquivo, index=False, sep=';', encoding='utf-8')
        elif formato == "xlsx":
            with pd.ExcelWriter(caminho_arquivo, engine='openpyxl') as writer:
```

```
                df.to_excel(writer, index=False, sheet_name='Relatorio Items')
                worksheet = writer.sheets['Relatorio Items']
                for col in worksheet.columns:
                    max_length = max(len(str(cell.value)) for cell in col)
                    adjusted_width = max(15, max_length + 2)
                    worksheet.column_dimensions[col[0].column_letter].width = adjusted_width
            else:
                raise ValueError("Formato inválido. Use: csv ou xlsx")
            return caminho_arquivo
        except Exception as e:
            raise HTTPException(status_code=500, detail=f"Erro ao salvar relatorio: {str(e)}")

    @staticmethod
    async def gerar_relatorio_quantidade_itens(
        session: AsyncSession,
        filtro_categoria: str = None,
        filtro_produto: str = None,
        formato: str = "csv"
    ):
        try:
            # 1. Buscar categorias (se houver filtro)
            categoria_ids = []
            if filtro_categoria:
                categorias = await CategoriaService.get_categorias_like(session, filtro_categoria)
                categoria_ids = [c.categoria_id for c in categorias]

            # 2. Buscar itens com filtros usando o ItemService
            itens = await ItemService.get_itens_filtrados(
                session,
                categoria_ids=categoria_ids,
                nome_produto=filtro_produto
            )

            # 3. Gerar DataFrame formatado
            df = formatar_dataframe_relatorio(itens)

            # 4. Exportar relatório
            caminho_arquivo = os.path.join(get_pasta_relatorios(), f"relatorio_quantidade_itens.{formato}")
            export_strategy = CSVExportStrategy() if formato == "csv" else XLSXExportStrategy()
            export_strategy.export(df, caminho_arquivo)

            return caminho_arquivo

        except HTTPException as e:
            raise e  # Repassa exceções HTTP específicas
        except Exception as e:
            raise HTTPException(
                status_code=500,
                detail=f"Erro ao gerar relatório: {str(e)}"
            )


    def formatar_dados_relatorio(itens: list[tuple]):
        return [{
            "ID_Categoria": item.categoria_id,
            "Nome_Categoria": nome_categoria,
            "Produto": item.nome_item_original.title(),
            "Quantidade": item.quantidade_item,
            "Data_Validade": item.data_validade_item.strftime('%d/%m/%Y') if item.data_validade_item else No
        } for item, nome_categoria in itens]

    def formatar_dataframe_relatorio(dados: list):
        if not dados:
```

```
            return pd.DataFrame(columns=["ID_Categoria", "Nome_Categoria", "Produto", "Quantidade", "Marca",

        df = pd.DataFrame(dados)
        if not df.empty:
            df['Produto'] = df['Produto'].str.title()
            if 'Marca' in df.columns:
                df['Marca'] = df['Marca'].str.title()
            if 'Data_Validade' in df.columns:
                df['Data_Validade'] = pd.to_datetime(df['Data_Validade']).dt.strftime('%d/%m/%Y')
        return df
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\utils\scheduler.py

```
# utils/scheduler.py

from apscheduler.schedulers.asyncio import AsyncIOScheduler
from services.alerta_service import AlertaService
from core.database import get_session_scheduler
from core.configs import settings
import os
from datetime import datetime, timedelta

scheduler = AsyncIOScheduler()

async def tarefa_diaria():
    async with get_session_scheduler() as db:
        print("Verificando validade e estoque dos itens...")
        await AlertaService.generate_daily_alerts(db)

async def tarefa_limpar_relatorios():
    print("Iniciando tarefa de limpeza de relatórios antigos...")
    pasta_relatorios = settings.PASTA_RELATORIOS
    dias_retencao = settings.REPORT_RETENTION_DAYS

    # Calcula a data de corte: arquivos mais antigos que esta data serão deletados
    data_corte = datetime.now() - timedelta(days=dias_retencao)

    for nome_arquivo in os.listdir(pasta_relatorios):
        caminho_arquivo = os.path.join(pasta_relatorios, nome_arquivo)

        # Verifica se é um arquivo (e não um diretório)
        if os.path.isfile(caminho_arquivo):
            try:
                # Obtém a data da última modificação do arquivo
                timestamp_modificacao = os.path.getmtime(caminho_arquivo)
                data_modificacao = datetime.fromtimestamp(timestamp_modificacao)

                if data_modificacao < data_corte:
                    os.remove(caminho_arquivo)
                    print(f"Relatório antigo removido: {nome_arquivo}")
            except Exception as e:
                print(f"Erro ao tentar remover o arquivo {nome_arquivo}: {e}")
    print("Tarefa de limpeza de relatórios concluída.")
```

Arquivo: C:\Users\Victor\Desktop\projeto_almoxarifado\sist_almoxarifado_ets\utils\websocket_endpoints.py

```python
# utils/websocket_endpoints.py

from fastapi import WebSocket, WebSocketDisconnect, APIRouter
from typing import Dict, List, Optional

websocket_router = APIRouter()

class ConnectionManager:
    def __init__(self):
        # Dicionário para mapear usuario_id para uma lista de WebSockets ativos
        self.active_connections: Dict[int, List[WebSocket]] = {}
        # Lista para conexões gerais (para alertas que não são específicos de usuário)
        self.general_connections: List[WebSocket] = []

    async def connect(self, websocket: WebSocket, user_id: int = None):
        await websocket.accept()
        if user_id:
            if user_id not in self.active_connections:
                self.active_connections[user_id] = []
            self.active_connections[user_id].append(websocket)
            print(f"WebSocket conectado para o usuário {user_id}. Total de conexões para este usuário: {
        else:
            self.general_connections.append(websocket)
            print(f"WebSocket conectado como conexão geral. Total: {len(self.general_connections)}")


    def disconnect(self, websocket: WebSocket, user_id: int = None):
        if user_id and user_id in self.active_connections:
            try:
                self.active_connections[user_id].remove(websocket)
                if not self.active_connections[user_id]:
                    del self.active_connections[user_id] # Remove a entrada se não houver mais conexões
                print(f"WebSocket desconectado para o usuário {user_id}.")
            except ValueError:
                pass # Conexão já removida ou não encontrada
        else:
            try:
                self.general_connections.remove(websocket)
                print("Conexão WebSocket geral desconectada.")
            except ValueError:
                pass # Conexão já removida ou não encontrada


    async def broadcast(self, message: dict):
        print(f"Attempting to broadcast message: {message}") # Log de depuração

        # Envia para conexões gerais
        for connection in list(self.general_connections): # Iterar sobre uma cópia para permitir remoçã
            try:
                await connection.send_json(message)
            except RuntimeError as e:
                print(f"Erro ao enviar mensagem para WebSocket geral: {e}. Desconectando...")
                self.general_connections.remove(connection)
            except WebSocketDisconnect:
                print("Conexão WebSocket geral já desconectada durante o broadcast.")
                self.general_connections.remove(connection)
            except Exception as e:
                print(f"Erro inesperado no broadcast para WebSocket geral: {e}")
                self.general_connections.remove(connection)

        #  Envia também para todas as conexões ativas de usuários específicos
        for user_id, connections in list(self.active_connections.items()):
            for connection in list(connections): # Iterar sobre uma cópia para permitir remoção
```

```
                try:
                    await connection.send_json(message)
                except RuntimeError as e:
                    print(f"Erro ao enviar mensagem para WebSocket do usuário {user_id} durante broadcas
                    connections.remove(connection)
                except WebSocketDisconnect:
                    print(f"Conexão WebSocket do usuário {user_id} já desconectada durante broadcast.")
                    connections.remove(connection)
                except Exception as e:
                    print(f"Erro inesperado no broadcast para WebSocket do usuário {user_id}: {e}")
                    connections.remove(connection)
            # Limpa listas de conexões de usuário vazias
            if not connections:
                del self.active_connections[user_id]


    async def send_to_user(self, user_id: int, message: dict):
        print(f"Attempting to send message to user {user_id}: {message}") # Log de depuração
        if user_id in self.active_connections:
            for connection in list(self.active_connections[user_id]):
                try:
                    await connection.send_json(message)
                except RuntimeError as e:
                    print(f"Erro ao enviar mensagem para WebSocket do usuário {user_id}: {e}. Desconecta
                    self.active_connections[user_id].remove(connection)
                except WebSocketDisconnect:
                    print(f"Conexão WebSocket do usuário {user_id} já desconectada.")
                    self.active_connections[user_id].remove(connection)
                except Exception as e:
                    print(f"Erro inesperado ao enviar para WebSocket do usuário {user_id}: {e}")
                    self.active_connections[user_id].remove(connection)
            # Limpa a lista de conexões de usuário vazia após iterar
            if not self.active_connections[user_id]:
                del self.active_connections[user_id]
        else:
            print(f"Nenhuma conexão WebSocket ativa para o usuário {user_id}.")


manager = ConnectionManager() # Instancia o ConnectionManager globalmente

@websocket_router.websocket("/ws/alerts")
async def websocket_endpoint(websocket: WebSocket):
    # O user_id será passado como um query parameter do frontend
    # Exemplo: ws://localhost:8082/api/almoxarifado/ws/alerts?user_id=123

    #  Obter user_id explicitamente dos query parameters
    user_id_str: Optional[str] = websocket.query_params.get("user_id")
    user_id: Optional[int] = None
    if user_id_str:
        try:
            user_id = int(user_id_str)
        except ValueError:
            print(f"Erro: user_id '{user_id_str}' não é um inteiro válido.")
            await websocket.close(code=1003, reason="Invalid user_id format")
            return

    await manager.connect(websocket, user_id)
    try:
        while True:
            # Mantém a conexão viva. Se não esperamos mensagens do cliente, ele simplesmente aguarda.
            await websocket.receive_text()
    except WebSocketDisconnect:
        print(f"Cliente WebSocket desconectado (user_id: {user_id}).")
```

```
    manager.disconnect(websocket, user_id)
except Exception as e:
    print(f"Erro inesperado no WebSocket (user_id: {user_id}): {e}")
    manager.disconnect(websocket, user_id)
```