Candidate Tasks

```
# app.py
import os
from datetime import datetime, date, timedelta
from typing import List, Optional

import asyncio
import aioredis
from fastapi import FastAPI, HTTPException, Depends, Query
from pydantic import BaseModel, EmailStr
from sqlalchemy import (Column, Date, DateTime, ForeignKey, Integer, SmallInteger,
                        String, Text, BigInteger, select, func, update)
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine
from sqlalchemy.orm import declarative_base, relationship, sessionmaker

DATABASE_URL = os.getenv("DATABASE_URL",
"postgresql+asyncpg://user:pass@localhost:5432/tasksdb")
REDIS_URL = os.getenv("REDIS_URL", "redis://localhost:6379/0")

Base = declarative_base()
# DB models
class User(Base):
    __tablename__ = "users"
    user_id = Column(BigInteger, primary_key=True, index=True)
    name = Column(String(255), nullable=False)
    email = Column(String(255), nullable=False, unique=True, index=True)
    registration_date = Column(DateTime(timezone=True), nullable=False, default=datetime.utcnow)
    tasks = relationship("Task", back_populates="user")
```

```
class Task(Base):
    __tablename__ = "tasks"
    task_id = Column(BigInteger, primary_key=True, index=True)
    title = Column(String(512), nullable=False)
    description = Column(Text)
    assigned_user_id = Column(BigInteger, ForeignKey("users.user_id", ondelete="CASCADE"), nullable=False, index=True)
    status = Column(SmallInteger, nullable=False, default=0)   # 0:pending,1:in-progress,2:completed
    priority = Column(SmallInteger, nullable=False, default=1) # 0:low,1:medium,2:high
    due_date = Column(Date)
    created_at = Column(DateTime(timezone=True), nullable=False, default=datetime.utcnow, index=True)
    updated_at = Column(DateTime(timezone=True), nullable=False, default=datetime.utcnow, onupdate=datetime.utcnow)
    version = Column(BigInteger, nullable=False, default=1)
    user = relationship("User", back_populates="tasks")


# Async DB session
engine = create_async_engine(DATABASE_URL, future=True, echo=False)
AsyncSessionLocal = sessionmaker(engine, class_=AsyncSession, expire_on_commit=False)


async def get_db():
    async with AsyncSessionLocal() as session:
        yield session


# Pydantic schemas
class UserCreate(BaseModel):
    name: str
    email: EmailStr


class UserOut(BaseModel):
    user_id: int
    name: str
    email: EmailStr
    registration_date: datetime
    class Config:
        orm_mode = True

class TaskCreate(BaseModel):
    title: str
    description: Optional[str] = None
    assigned_user_id: int
    priority: int = 1
    due_date: Optional[date] = None

class TaskUpdate(BaseModel):
    title: Optional[str] = None
    description: Optional[str] = None
    assigned_user_id: Optional[int] = None
    status: Optional[int] = None
    priority: Optional[int] = None
    due_date: Optional[date] = None
    version: int  # required for optimistic locking

class TaskOut(BaseModel):
    task_id: int
    title: str
    description: Optional[str]
    assigned_user_id: int
    status: int
    priority: int
    due_date: Optional[date]
    created_at: datetime
    updated_at: datetime
    version: int
    class Config:
        orm_mode = True

# App + Redis
app = FastAPI(title="Humanized AI - Task Service")
redis = None

@app.on_event("startup")
async def startup():
    global redis
    redis = await aioredis.from_url(REDIS_URL, encoding="utf-8", decode_responses=True)

@app.on_event("shutdown")
async def shutdown():
    global redis
    if redis:
        await redis.close()

# User APIs
@app.post("/api/register", response_model=UserOut)
async def register(user_in: UserCreate, db: AsyncSession = Depends(get_db)):
    # simple register; no auth implementation for brevity
    existing = await db.execute(select(User).where(User.email == user_in.email))
    if existing.scalars().first():
        raise HTTPException(status_code=400, detail="Email already registered")
    new = User(name=user_in.name, email=user_in.email)
    db.add(new)
    await db.commit()
    await db.refresh(new)
    return new


@app.get("/api/users/{user_id}/tasks", response_model=List[TaskOut])
async def get_user_tasks(user_id: int, limit: int = 20, offset: int = 0, db: AsyncSession = Depends(get_db)):
    q = select(Task).where(Task.assigned_user_id == user_id).order_by(Task.due_date.nulls_last(), Task.created_at.desc()).limit(limit).offset(offset)
    res = await db.execute(q)
    return res.scalars().all()


# Task APIs
@app.post("/api/tasks", response_model=TaskOut)
async def create_task(payload: TaskCreate, db: AsyncSession = Depends(get_db)):
    # transactionally create task
    new = Task(
        title=payload.title,
        description=payload.description,
        assigned_user_id=payload.assigned_user_id,
        status=0,
        priority=payload.priority,
        due_date=payload.due_date
    )
    db.add(new)
    await db.commit()
    await db.refresh(new)
    # invalidate leaderboard cache
    await redis.delete("leaderboard:top")
    return new


@app.get("/api/tasks", response_model=List[TaskOut])
async def list_tasks(
    status: Optional[int] = Query(None),
    priority: Optional[int] = Query(None),
    due_from: Optional[date] = Query(None),
    due_to: Optional[date] = Query(None),
    assigned_user_id: Optional[int] = Query(None),
    limit: int = 20, page_token: Optional[int] = None, db: AsyncSession = Depends(get_db)
):
    # keyset pagination: page_token is last task_id seen
    q = select(Task)
    if status is not None:
        q = q.where(Task.status == status)
    if priority is not None:
        q = q.where(Task.priority == priority)
    if due_from is not None:
        q = q.where(Task.due_date >= due_from)
    if due_to is not None:
        q = q.where(Task.due_date <= due_to)
    if assigned_user_id is not None:
        q = q.where(Task.assigned_user_id == assigned_user_id)
    q = q.order_by(Task.task_id.asc()).limit(limit)
    if page_token:
        q = q.where(Task.task_id > page_token)
    res = await db.execute(q)
    return res.scalars().all()

@app.put("/api/tasks/{task_id}", response_model=TaskOut)
async def update_task(task_id: int, payload: TaskUpdate, db: AsyncSession = Depends(get_db)):
    # optimistic locking: update only if version matches
    stmt = (
        update(Task)
        .where(Task.task_id == task_id)
        .where(Task.version == payload.version)
        .values(
            title = payload.title if payload.title is not None else Task.title,
            description = payload.description if payload.description is not None else Task.description,
            assigned_user_id = payload.assigned_user_id if payload.assigned_user_id is not None else
Task.assigned_user_id,
            status = payload.status if payload.status is not None else Task.status,
            priority = payload.priority if payload.priority is not None else Task.priority,
            due_date = payload.due_date if payload.due_date is not None else Task.due_date,
            version = Task.version + 1,
            updated_at = datetime.utcnow()
        )
        .execution_options(synchronize_session="fetch")
    )
    res = await db.execute(stmt)
    if res.rowcount == 0:
        raise HTTPException(status_code=409, detail="Version conflict — reload and retry")
    await db.commit()
    # refresh and return
    refreshed = await db.execute(select(Task).where(Task.task_id == task_id))
    task = refreshed.scalars().first()
    # invalidate caches if status changed to completed (simple heuristic)
    if payload.status == 2:
        await redis.delete("leaderboard:top")
    return task


# Analytics / Leaderboard
@app.get("/api/leaderboard")
async def leaderboard(limit: int = 10, db: AsyncSession = Depends(get_db)):
    # try cache
    cached = await redis.get("leaderboard:top")
    if cached:
```

```
    return {"source": "cache", "data": cached}
# compute: top users by count of completed tasks in last 30 days
since = datetime.utcnow() - timedelta(days=30)
q = (
    select(User.user_id, User.name, func.count(Task.task_id).label("completed"))
    .join(Task, Task.assigned_user_id == User.user_id)
    .where(Task.status == 2)
    .where(Task.updated_at >= since)
    .group_by(User.user_id, User.name)
    .order_by(func.count(Task.task_id).desc())
    .limit(limit)
)
res = await db.execute(q)
rows = [{"user_id": r.user_id, "name": r.name, "completed": int(r.completed)} for r in res]
# store cache for short time
await redis.set("leaderboard:top", str(rows), ex=60)  # 60s cache
return {"source": "db", "data": rows}
```