```python
import asyncio
import json
import time
from datetime import datetime
from typing import Any, Dict, List, Optional

import aiohttp
import backoff
import tweepy
from fastapi import (
    BackgroundTasks,
    FastAPI,
    HTTPException,
    Response,
    status,
)
from fastapi.middleware.cors import CORSMiddleware
from loguru import logger
from pydantic import BaseModel, Field

from mcs.main import MedicalCoderSwarm

# Configure logging
logger.add(
    "twitter_bot.log",
    rotation="500 MB",
    retention="10 days",
    level="INFO",
    backtrace=True,
    diagnose=True,
)

# FastAPI instance with CORS
app = FastAPI(title="Twitter Bot API", version="2.0.0")
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)


# Configuration class
class TwitterConfig:
    def __init__(self):
        self.API_KEY = "YOUR_API_KEY"
        self.API_SECRET = "YOUR_API_SECRET"
        self.ACCESS_TOKEN = "YOUR_ACCESS_TOKEN"
        self.ACCESS_SECRET = "YOUR_ACCESS_SECRET"
        self.BEARER_TOKEN = "YOUR_BEARER_TOKEN"
        self.POLL_INTERVAL = 60  # seconds
        self.MAX_RETRIES = 3
        self.RATE_LIMIT_WINDOW = 900  # 15 minutes in seconds
        self.MAX_REQUESTS_PER_WINDOW = (
            180  # Twitter's rate limit for most endpoints
        )


config = TwitterConfig()


# Pydantic models
class MentionRequest(BaseModel):
    response_template: str = Field(..., min_length=1, max_length=280)
    keywords: Optional[List[str]] = Field(default=[])


class DMRequest(BaseModel):
    user_id: str
    message: str = Field(..., min_length=1, max_length=1000)


class Task(BaseModel):
    tweet_id: str
    user: str
    content: str
    response: str
    created_at: datetime = Field(default_factory=datetime.now)

    status: str = Field(default="pending")




# In-memory storage (replace with database in production)

class Storage:

    def __init__(self):

        self.tasks: List[Task] = []

        self.rate_limits: Dict[str, List[float]] = {}

        self.last_mention_id: Optional[str] = None


    def add_task(self, task: Task):

        self.tasks.append(task)


    def get_tasks(self) -> List[Task]:

        return self.tasks


    def check_rate_limit(self, endpoint: str) -> bool:

        now = time.time()

        if endpoint not in self.rate_limits:

            self.rate_limits[endpoint] = []


        # Remove old timestamps

        self.rate_limits[endpoint] = [

            ts

            for ts in self.rate_limits[endpoint]
            if now - ts < config.RATE_LIMIT_WINDOW
        ]

        return (
            len(self.rate_limits[endpoint])
            < config.MAX_REQUESTS_PER_WINDOW
        )

    def add_rate_limit_timestamp(self, endpoint: str):
        if endpoint not in self.rate_limits:
            self.rate_limits[endpoint] = []
        self.rate_limits[endpoint].append(time.time())


storage = Storage()


# Twitter client class with retry logic
class TwitterClient:
    def __init__(self):
        self.client = tweepy.Client(
            bearer_token=config.BEARER_TOKEN,
            consumer_key=config.API_KEY,
            consumer_secret=config.API_SECRET,
            access_token=config.ACCESS_TOKEN,
            access_token_secret=config.ACCESS_SECRET,
            wait_on_rate_limit=True,
        )


    @backoff.on_exception(
        backoff.expo,
        (tweepy.TweepyException, aiohttp.ClientError),
        max_tries=config.MAX_RETRIES,
    )
    async def send_dm(self, user_id: str, message: str) -> bool:
        """Send a direct message with retry logic."""
        try:
            if not storage.check_rate_limit("dm"):
                raise HTTPException(
                    status_code=429,
                    detail="Rate limit exceeded for DM endpoint",
                )

            self.client.create_direct_message(
                participant_id=user_id, text=message
            )
            storage.add_rate_limit_timestamp("dm")
            logger.info(f"Successfully sent DM to user {user_id}")
            return True

        except Exception as e:
            logger.error(
                f"Failed to send DM to user {user_id}: {str(e)}"
            )
            raise

    @backoff.on_exception(
        backoff.expo,
        (tweepy.TweepyException, aiohttp.ClientError),
        max_tries=config.MAX_RETRIES,
    )
    async def reply_to_tweet(
        self, tweet_id: str, user: str, message: str
    ) -> bool:
        """Reply to a tweet with retry logic."""
        try:
            if not storage.check_rate_limit("tweet"):
                raise HTTPException(
                    status_code=429,
                    detail="Rate limit exceeded for tweet endpoint",
                )

            self.client.create_tweet(
                text=f"@{user} {message}",
                in_reply_to_tweet_id=tweet_id,
            )
            storage.add_rate_limit_timestamp("tweet")
            logger.info(f"Successfully replied to tweet {tweet_id}")
            return True

        except Exception as e:
            logger.error(
                f"Failed to reply to tweet {tweet_id}: {str(e)}"
            )
            raise

    @backoff.on_exception(
        backoff.expo,
        (tweepy.TweepyException, aiohttp.ClientError),
        max_tries=config.MAX_RETRIES,
    )
    async def get_mentions(self) -> List[Dict[str, Any]]:
        """Get mentions with retry logic."""
        try:
            if not storage.check_rate_limit("mentions"):
                raise HTTPException(
                    status_code=429,
                    detail="Rate limit exceeded for mentions endpoint",
                )

            mentions = self.client.get_mentions(
                since_id=storage.last_mention_id,
                tweet_fields=["created_at", "text"],
                user_fields=["username"],
            )

            storage.add_rate_limit_timestamp("mentions")

            if mentions.data:
                storage.last_mention_id = mentions.data[0].id

            return mentions.data or []

        except Exception as e:
            logger.error(f"Failed to fetch mentions: {str(e)}")
            raise


twitter_client = TwitterClient()


# Medical coder processing
async def process_medical_coding(tweet_id: str, content: str) -> str:
    """Process medical coding with error handling."""
    try:
        swarm = MedicalCoderSwarm(
            patient_id=tweet_id, max_loops=1, patient_documentation=""
        )
        response_data = swarm.run(task=content)
        logger.info(f"Medical coding completed for tweet {tweet_id}")
        return response_data

    except Exception as e:
        logger.error(
            f"Medical coding failed for tweet {tweet_id}: {str(e)}"
        )
        return "I apologize, but I couldn't process your request at this time."


# Mention processing
async def process_mention(
    mention: Dict[str, Any],
    response_template: str,
    keywords: List[str],
) -> None:
    """Process a single mention."""
    tweet_id = mention.id
    user = mention.author.username
    content = mention.text

    # Check if mention contains any keywords (if specified)
    if keywords and not any(
        keyword.lower() in content.lower() for keyword in keywords
    ):
        logger.info(
            f"Tweet {tweet_id} doesn't contain any keywords, skipping"
        )
        return

    try:
        # Process medical coding
        response_data = await process_medical_coding(
            tweet_id, content
        )

        # Create and store task
        task = Task(
            tweet_id=str(tweet_id),
            user=user,
            content=content,
            response=response_data,
            status="completed",
        )
        storage.add_task(task)

        # Send reply
        await twitter_client.reply_to_tweet(
            tweet_id=str(tweet_id),
            user=user,
            message=f"{response_template}\n\n{response_data}",
        )
    except Exception as e:
        logger.error(
            f"Failed to process mention {tweet_id}: {str(e)}"
        )
        # Store failed task
        task = Task(
            tweet_id=str(tweet_id),
            user=user,
            content=content,
            response=str(e),
            status="failed",
        )
        storage.add_task(task)


# Background mention polling
async def poll_mentions(response_template: str, keywords: List[str]):
    """Poll mentions continuously."""
    while True:
        try:
            mentions = await twitter_client.get_mentions()

            for mention in mentions:
                await process_mention(
                    mention, response_template, keywords
                )
            await asyncio.sleep(config.POLL_INTERVAL)

        except Exception as e:
            logger.error(f"Error in mention polling: {str(e)}")
            await asyncio.sleep(config.POLL_INTERVAL)


# API endpoints
@app.post("/start-polling")
async def start_polling(
    request: MentionRequest, background_tasks: BackgroundTasks
):
    """Start polling mentions."""
    try:
        background_tasks.add_task(
            poll_mentions, request.response_template, request.keywords
        )
        return {"status": "Polling started successfully"}
    except Exception as e:
        logger.error(f"Failed to start polling: {str(e)}")
        raise HTTPException(
            status_code=500, detail="Failed to start polling"
        )
```

```
@app.post("/send-dm")
async def send_dm(request: DMRequest):
    """Send a direct message."""
    try:
        await twitter_client.send_dm(request.user_id, request.message)
        return {"status": "DM sent successfully"}
    except Exception as e:
        logger.error(f"Failed to send DM: {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))


@app.get("/tasks")
async def get_tasks(
    response: Response, limit: int = 100, offset: int = 0
):
    """Get tasks with pagination."""
    try:
        tasks = storage.get_tasks()
        total = len(tasks)

        # Add pagination headers
        response.headers["X-Total-Count"] = str(total)
        response.headers["X-Limit"] = str(limit)
        response.headers["X-Offset"] = str(offset)

        return tasks[offset : offset + limit]
    except Exception as e:
        logger.error(f"Failed to fetch tasks: {str(e)}")
        raise HTTPException(
            status_code=500, detail="Failed to fetch tasks"
        )


@app.get("/health")
async def health_check():
    """Health check endpoint."""
    try:
        # Verify Twitter credentials
        await twitter_client.get_mentions()
        return {"status": "healthy"}
    except Exception as e:
        logger.error(f"Health check failed: {str(e)}")
        return Response(
            content=json.dumps(
                {"status": "unhealthy", "error": str(e)}
            ),
            status_code=status.HTTP_503_SERVICE_UNAVAILABLE,
        )


if __name__ == "__main__":
    import uvicorn
```

```
logger.info("Starting Twitter Bot API server...")

uvicorn.run(

    "main:app", host="0.0.0.0", port=8000, reload=True, workers=4

)
```