```
import json
from datetime import datetime
from typing import Dict, List


import discord
from discord.ext import commands
from fastapi import (
    BackgroundTasks,
    Depends,
    FastAPI,
    HTTPException,
    Response,
    status,
)
from fastapi.middleware.cors import CORSMiddleware
from loguru import logger
from sqlalchemy import (
    Column,
    DateTime,
    ForeignKey,
    Integer,
    String,
    Text,
)
from sqlalchemy.ext.asyncio import (
    AsyncSession,
```

```
    async_sessionmaker,
    create_async_engine,
)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.future import select
from sqlalchemy.orm import relationship

from mcs.main import MedicalCoderSwarm

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

# FastAPI instance with CORS
app = FastAPI(title="Discord Medical Bot API", version="1.0.0")
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
```

```
    allow_headers=["*"],
)



# Configuration
class BotConfig:
    def __init__(self):
        self.DISCORD_TOKEN = "YOUR_DISCORD_BOT_TOKEN"
        self.DATABASE_URL = "sqlite+aiosqlite:///./discord_bot.db"
        self.MAX_HISTORY_LENGTH = 100
        self.MAX_RETRIES = 3
        self.RATE_LIMIT_WINDOW = 60
        self.MAX_REQUESTS_PER_MINUTE = 30
        self.COMMAND_PREFIX = "!"



config = BotConfig()

# Database setup
Base = declarative_base()



class Conversation(Base):
    __tablename__ = "conversations"

    id = Column(Integer, primary_key=True)
```

```
    user_id = Column(String(100), unique=True, index=True)
    messages = relationship(
        "Message",
        back_populates="conversation",
        cascade="all, delete-orphan",
    )


class Message(Base):
    __tablename__ = "messages"

    id = Column(Integer, primary_key=True)
    conversation_id = Column(Integer, ForeignKey("conversations.id"))
    content = Column(Text)
    timestamp = Column(DateTime, default=datetime.utcnow)
    role = Column(String(50))  # 'user' or 'assistant'

    conversation = relationship(
        "Conversation", back_populates="messages"
    )


# Create async engine
engine = create_async_engine(config.DATABASE_URL, echo=True)
AsyncSessionLocal = async_sessionmaker(engine, expire_on_commit=False)
```

```
async def init_db():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)



# Dependency for database sessions
async def get_db():
    async with AsyncSessionLocal() as session:
        try:
            yield session
        finally:
            await session.close()



# Database operations
class DatabaseOps:
    @staticmethod
    async def store_message(
        db: AsyncSession, user_id: str, content: str, role: str
    ):
        """Store a message in conversation history."""
        stmt = select(Conversation).where(
            Conversation.user_id == str(user_id)
        )
        result = await db.execute(stmt)
```

```
conversation = result.scalar_one_or_none()

if not conversation:
    conversation = Conversation(user_id=str(user_id))
    db.add(conversation)
    await db.flush()

message = Message(
    conversation_id=conversation.id,
    content=content,
    role=role,
)
db.add(message)

# Maintain message limit
stmt = (
    select(Message)
    .where(Message.conversation_id == conversation.id)
    .order_by(Message.timestamp)
)
result = await db.execute(stmt)
messages = result.scalars().all()

if len(messages) > config.MAX_HISTORY_LENGTH:
    for msg in messages[: -config.MAX_HISTORY_LENGTH]:
        await db.delete(msg)
```

```
    await db.commit()


@staticmethod
async def get_conversation_history(
    db: AsyncSession, user_id: str
) -> List[Dict]:
    """Get conversation history for a user."""
    stmt = select(Conversation).where(
        Conversation.user_id == str(user_id)
    )
    result = await db.execute(stmt)
    conversation = result.scalar_one_or_none()

    if not conversation:
        return []

    stmt = (
        select(Message)
        .where(Message.conversation_id == conversation.id)
        .order_by(Message.timestamp)
    )
    result = await db.execute(stmt)
    messages = result.scalars().all()

    return [
```

```
            {
                "content": msg.content,
                "timestamp": msg.timestamp,
                "role": msg.role,
            }
            for msg in messages
        ]

    @staticmethod
    async def clear_history(db: AsyncSession, user_id: str):
        """Clear conversation history for a user."""
        stmt = select(Conversation).where(
            Conversation.user_id == str(user_id)
        )
        result = await db.execute(stmt)
        conversation = result.scalar_one_or_none()

        if conversation:
            await db.delete(conversation)
            await db.commit()


# Medical Coder Swarm with context
class ContextAwareMedicalSwarm:
    def __init__(self, user_id: str):
        self.user_id = user_id
```

```
    self.swarm = MedicalCoderSwarm(
        patient_id=user_id, max_loops=1, patient_documentation=""
    )


async def process_with_context(
    self, current_message: str, db: AsyncSession
) -> str:
    """Process message with conversation context."""
    try:
        history = await DatabaseOps.get_conversation_history(
            db, self.user_id
        )

        context = "\n".join(
            [
                f"{msg['role']}: {msg['content']}"
                for msg in history
            ]
        )

        full_context = f"{context}\nUser: {current_message}"

        response = self.swarm.run(
            task=current_message, context=full_context
        )
```

```
            return response

        except Exception as e:
            logger.error(
                f"Swarm processing error for user {self.user_id}: {str(e)}"
            )
            return "I apologize, but I couldn't process your request at this time."



# Discord bot class
class MedicalBot(commands.Bot):
    def __init__(self):
        intents = discord.Intents.default()
        intents.message_content = True
        intents.dm_messages = True

        super().__init__(
            command_prefix=commands.when_mentioned_or(
                config.COMMAND_PREFIX
            ),
            intents=intents,
        )

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

```
async def setup_hook(self):
    await self.tree.sync()


def check_rate_limit(self, user_id: str) -> bool:
    """Check if user has exceeded rate limit."""
    now = datetime.now().timestamp()
    if user_id not in self.rate_limits:
        self.rate_limits[user_id] = []

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

    return (
        len(self.rate_limits[user_id])
        < config.MAX_REQUESTS_PER_MINUTE
    )


def add_rate_limit_timestamp(self, user_id: str):
    """Add timestamp for rate limiting."""
    if user_id not in self.rate_limits:
        self.rate_limits[user_id] = []
    self.rate_limits[user_id].append(datetime.now().timestamp())
```

```
# Create bot instance
bot = MedicalBot()



# Command handlers
@bot.tree.command(
    name="help", description="Show available commands and usage"
)
async def help_command(interaction: discord.Interaction):
    """Handle help command."""
    help_embed = discord.Embed(
        title="Medical Coding Assistant Help",
        description="Here are the available commands:",
        color=discord.Color.blue(),
    )

    help_embed.add_field(
        name="/help", value="Show this help message", inline=False
    )

    help_embed.add_field(
        name="/analyze <text>",
        value="Analyze medical text for coding",
        inline=False,
    )
```

```
    help_embed.add_field(
        name="/clear",
        value="Clear your conversation history",
        inline=False,
    )

    help_embed.add_field(
        name="DM Functionality",
        value="You can also DM me directly for a natural conversation with memory!",
        inline=False,
    )

    await interaction.response.send_message(embed=help_embed)


@bot.tree.command(
    name="analyze", description="Analyze medical text for coding"
)
async def analyze_command(
    interaction: discord.Interaction, text: str
):
    """Handle analyze command."""
    user_id = str(interaction.user.id)

    if not bot.check_rate_limit(user_id):
```

```
        await interaction.response.send_message(
            "You're sending requests too quickly. Please wait a moment.",
            ephemeral=True,
        )
        return

    async with bot.db_session() as db:
        try:
            # Store user message
            await DatabaseOps.store_message(db, user_id, text, "user")

            # Process with swarm
            swarm = ContextAwareMedicalSwarm(user_id)
            response = await swarm.process_with_context(text, db)

            # Store bot response
            await DatabaseOps.store_message(
                db, user_id, response, "assistant"
            )

            # Send response
            await interaction.response.send_message(response)
            bot.add_rate_limit_timestamp(user_id)

        except Exception as e:
            logger.error(
```

```
                f"Error processing analyze command: {str(e)}"
            )
            await interaction.response.send_message(
                "I encountered an error processing your request. Please try again later.",
                ephemeral=True,
            )



@bot.tree.command(
    name="clear", description="Clear your conversation history"
)
async def clear_command(interaction: discord.Interaction):
    """Handle clear command."""
    async with bot.db_session() as db:
        try:
            await DatabaseOps.clear_history(
                db, str(interaction.user.id)
            )
            await interaction.response.send_message(
                "Your conversation history has been cleared! ",
                ephemeral=True,
            )
        except Exception as e:
            logger.error(f"Error clearing history: {str(e)}")
            await interaction.response.send_message(
                "Failed to clear conversation history. Please try again later.",
```

```
            ephemeral=True,
        )



# DM handler
@bot.event
async def on_message(message: discord.Message):
    """Handle direct messages."""
    # Ignore bot messages and non-DM messages
    if message.author.bot or not isinstance(
        message.channel, discord.DMChannel
    ):
        return

    user_id = str(message.author.id)

    if not bot.check_rate_limit(user_id):
        await message.reply(
            "You're sending messages too quickly. Please wait a moment."
        )
        return

    async with bot.db_session() as db:
        try:
            # Store user message
            await DatabaseOps.store_message(
```

```
            db, user_id, message.content, "user"
        )

        # Process with swarm
        swarm = ContextAwareMedicalSwarm(user_id)
        response = await swarm.process_with_context(
            message.content, db
        )

        # Store bot response
        await DatabaseOps.store_message(
            db, user_id, response, "assistant"
        )

        # Send response
        await message.reply(response)
        bot.add_rate_limit_timestamp(user_id)

    except Exception as e:
        logger.error(f"Error processing DM: {str(e)}")
        await message.reply(
            "I encountered an error processing your message. "
            "Please try again later."
        )
```

```
# FastAPI endpoints
@app.post("/start")
async def start_bot(background_tasks: BackgroundTasks):
    """Start the Discord bot."""
    try:
        # Initialize database
        await init_db()

        # Start bot
        background_tasks.add_task(bot.start, config.DISCORD_TOKEN)
        return {"status": "Bot started successfully"}
    except Exception as e:
        logger.error(f"Failed to start bot: {str(e)}")
        raise HTTPException(
            status_code=500, detail="Failed to start bot"
        )


@app.get("/conversations/{user_id}")
async def get_conversation(
    user_id: str, db: AsyncSession = Depends(get_db)
):
    """Get conversation history for a user."""
    try:
        history = await DatabaseOps.get_conversation_history(
            db, user_id
```

```
        )
        return {"user_id": user_id, "messages": history}
    except Exception as e:
        logger.error(f"Failed to fetch conversation: {str(e)}")
        raise HTTPException(
            status_code=500, detail="Failed to fetch conversation"
        )


@app.delete("/conversations/{user_id}")
async def clear_conversation(
    user_id: str, db: AsyncSession = Depends(get_db)
):
    """Clear conversation history for a user."""
    try:
        await DatabaseOps.clear_history(db, user_id)
        return {"status": "Conversation cleared successfully"}
    except Exception as e:
        logger.error(f"Failed to clear conversation: {str(e)}")
        raise HTTPException(
            status_code=500, detail="Failed to clear conversation"
        )


@app.get("/health")
async def health_check(db: AsyncSession = Depends(get_db)):
```

```
    """Health check endpoint."""
    try:
        # Check database connection
        await db.execute("SELECT 1")

        # Check Discord bot connection
        if not bot.is_ready():
            raise Exception("Discord bot is not connected")

        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 Discord Bot API server...")
    uvicorn.run(
        "main:app",
```

```
    host="0.0.0.0",
    port=8000,
    reload=True,
    workers=1,  # Use 1 worker for Discord bot
)
```