```
import asyncio
import json
from contextlib import asynccontextmanager
from datetime import datetime
import os
from typing import Dict, List

import aiohttp
import backoff
from fastapi import (
    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 telegram import Bot, Update
from telegram.ext import (
    Application,
    CommandHandler,
    ContextTypes,
    MessageHandler,
    filters,
)

from mcs import MedicalCoderSwarm
from dotenv import load_dotenv

load_dotenv()



async def verify_bot_token():
    """Verify the bot token is valid"""
    try:
```

```
        bot = Bot(config.TELEGRAM_TOKEN)
        bot_info = await bot.get_me()
        logger.info(f"Bot verified: @{bot_info.username}")
        return True
    except Exception as e:
        logger.error(f"Bot token verification failed: {str(e)}")
        return False


@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup: Initialize the database and start the Telegram bot
    # Verify token first
    if not await verify_bot_token():
        raise RuntimeError("Invalid bot token")

    await init_db()
    await bot_handler.bot.initialize()
    await bot_handler.bot.start()

    yield  # Server is running

    # Shutdown: Stop the Telegram bot
    await bot_handler.bot.stop()
    await bot_handler.bot.shutdown()
```

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

# FastAPI instance with CORS
app = FastAPI(
    title="Telegram Medical Bot API",
    version="1.0.0",
    lifespan=lifespan,
    debug=True,
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

```
# Configuration
class BotConfig:
    def __init__(self):
        self.TELEGRAM_TOKEN = os.getenv("TELEGRAM_API_KEY")
        self.DATABASE_URL = "sqlite+aiosqlite:///./telegram_bot.db"
        self.MAX_HISTORY_LENGTH = 50
        self.MAX_RETRIES = 3
        self.RATE_LIMIT_WINDOW = 60  # seconds
        self.MAX_REQUESTS_PER_MINUTE = 30


config = BotConfig()

# Database setup
Base = declarative_base()


class Conversation(Base):
    __tablename__ = "conversations"

    id = Column(Integer, primary_key=True)
    chat_id = Column(Integer, 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"))
    text = Column(Text)
    timestamp = Column(DateTime, default=datetime.utcnow)
    role = Column(String(50))

    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
class DatabaseOps:
    @staticmethod
    async def store_message(
        db: AsyncSession, chat_id: int, text: str, role: str
    ):
        """Store a message in the conversation history."""
        # Get or create conversation
        stmt = select(Conversation).where(
            Conversation.chat_id == chat_id
        )
        result = await db.execute(stmt)
        conversation = result.scalar_one_or_none()
```

```
if not conversation:
    conversation = Conversation(chat_id=chat_id)
    db.add(conversation)
    await db.flush()

# Add new message
message = Message(
    conversation_id=conversation.id,
    text=text,
    role=role,
    timestamp=datetime.utcnow(),
)
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:
    # Remove oldest messages
    for msg in messages[: -config.MAX_HISTORY_LENGTH]:
```

```
            await db.delete(msg)

    await db.commit()

@staticmethod
async def get_conversation_history(
    db: AsyncSession, chat_id: int
) -> List[Dict]:
    """Retrieve conversation history for a chat."""
    stmt = select(Conversation).where(
        Conversation.chat_id == chat_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 [
            {
                "text": msg.text,
                "timestamp": msg.timestamp,
                "role": msg.role,
            }
            for msg in messages
        ]

    @staticmethod
    async def clear_history(db: AsyncSession, chat_id: int):
        """Clear conversation history for a chat."""
        stmt = select(Conversation).where(
            Conversation.chat_id == chat_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, chat_id: int):
```

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

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

        # Format context for the swarm
        context = "\n".join(
            [f"{msg['role']}: {msg['text']}" for msg in history]
        )

        # Add current message to context
        full_context = f"{context}\nUser: {current_message}"

        # Process with swarm
```

```
            response = self.swarm.run(
                task=full_context + "\n" + current_message,
            )

            return response

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


# Telegram bot handler class
class TelegramBotHandler:
    def __init__(self):
        self.bot = (
            Application.builder().token(config.TELEGRAM_TOKEN).build()
        )
        self.setup_handlers()
        self.rate_limits: Dict[int, List[float]] = {}

    def setup_handlers(self):
        """Set up message handlers."""
        self.bot.add_handler(
            CommandHandler("start", self.start_command)
```

```
    )
    self.bot.add_handler(
        CommandHandler("help", self.help_command)
    )
    self.bot.add_handler(
        CommandHandler("clear", self.clear_command)
    )
    self.bot.add_handler(
        MessageHandler(
            filters.TEXT & ~filters.COMMAND, self.handle_message
        )
    )

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

    # Remove old timestamps
    self.rate_limits[chat_id] = [
        ts
        for ts in self.rate_limits[chat_id]
        if now - ts < config.RATE_LIMIT_WINDOW
    ]
```

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

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

async def start_command(
    self, update: Update, context: ContextTypes.DEFAULT_TYPE
):
    """Handle /start command."""
    welcome_message = (
        "Welcome to the Medical Coding Assistant! \n\n"
        "I can help you with medical coding questions and maintain context "
        "of our entire conversation. Feel free to ask any questions!\n\n"
        "Use /help to see available commands."
    )
    await update.message.reply_text(welcome_message)

async def help_command(
    self, update: Update, context: ContextTypes.DEFAULT_TYPE
):
```

```
    """Handle /help command."""
    help_message = (
        "Available commands:\n"
        "/start - Start the bot\n"
        "/help - Show this help message\n"
        "/clear - Clear conversation history\n\n"
        "Simply send me any message to get medical coding assistance!"
    )
    await update.message.reply_text(help_message)

async def clear_command(
    self, update: Update, context: ContextTypes.DEFAULT_TYPE
):
    """Handle /clear command."""
    async with AsyncSessionLocal() as db:
        chat_id = update.effective_chat.id
        await DatabaseOps.clear_history(db, chat_id)
        await update.message.reply_text(
            "Conversation history has been cleared! "
        )

@backoff.on_exception(
    backoff.expo,
    (aiohttp.ClientError, asyncio.TimeoutError),
    max_tries=config.MAX_RETRIES,
)
```

```
async def handle_message(
    self, update: Update, context: ContextTypes.DEFAULT_TYPE
):
    """Handle incoming messages."""
    chat_id = update.effective_chat.id
    user_message = update.message.text

    # Check rate limit
    if not self.check_rate_limit(chat_id):
        await update.message.reply_text(
            "You're sending messages too quickly. Please wait a moment."
        )
        return

    async with AsyncSessionLocal() as db:
        try:
            # Store user message
            await DatabaseOps.store_message(
                db, chat_id, user_message, "user"
            )

            # Process with swarm
            swarm = ContextAwareMedicalSwarm(chat_id)
            response = await swarm.process_with_context(
                user_message, db
            )
```

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

            # Send response
            await update.message.reply_text(response)
            self.add_rate_limit_timestamp(chat_id)

        except Exception:
            logger.exception(
                f"Detailed error processing message for chat {chat_id}"
            )
            await update.message.reply_text(
                "I'm sorry, but I encountered an error processing your message. "
                "Please try again later."
            )


# Initialize bot handler
bot_handler = TelegramBotHandler()


# API endpoints
@app.post("/start")
```

```
async def start_bot():
    """Start the Telegram bot."""
    try:
        # The bot is already initialized in lifespan
        # Just start polling
        await app.state.bot_handler.bot.run_polling(
            allowed_updates=["message"]
        )
        return {"status": "Bot started successfully"}
    except Exception as e:
        logger.error(f"Failed to start bot: {str(e)}")
        raise HTTPException(
            status_code=500, detail=f"Failed to start bot: {str(e)}"
        )



@app.get("/conversations/{chat_id}")
async def get_conversation(
    chat_id: int, db: AsyncSession = Depends(get_db)
):
    """Get conversation history for a chat."""
    try:
        history = await DatabaseOps.get_conversation_history(
            db, chat_id
        )
        return {"chat_id": chat_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/{chat_id}")
async def clear_conversation(
    chat_id: int, db: AsyncSession = Depends(get_db)
):
    """Clear conversation history for a chat."""
    try:
        await DatabaseOps.clear_history(db, chat_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 Telegram bot
        bot = Bot(config.TELEGRAM_TOKEN)
        await bot.get_me()

        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

    print("Starting server...")  # Visual confirmation
    uvicorn.run(
        app,
        host="127.0.0.1",  # Using localhost explicitly
        port=8002,
```

```
    workers=1,
    log_level="debug",
    use_colors=True,
    reload=True,  # Enables auto-reload for development
)
```