```
import time

import json

import os


from fastapi import FastAPI, HTTPException, Request

from fastapi.middleware.cors import CORSMiddleware

from swarms import Agent, OpenAIChat

from swarms.utils.loguru_logger import logger

from swarms_cloud.schema.cog_vlm_schemas import ChatCompletionResponse, UsageInfo

from swarms_cloud.schema.agent_api_schemas import (

    AgentInput,

    AgentOutput,

    ModelList,

    ModelSchema,

    AllAgentsSchema,

    AgentCreationOutput,

)

from swarms_memory import ChromaDB

from swarms.models.tiktoken_wrapper import TikTokenizer


logger.info("Starting the agent API server...")


llm = OpenAIChat(

    max_tokens=4000,

    model_name="gpt-4o",

    api_key=os.getenv("OPENAI_API_KEY"),
)


# Create a FastAPI app
app = FastAPI(
    debug=True,
    title="Swarm Agent API",
    version="0.1.0",
)


# Load the middleware to handle CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)



@app.get("/v1/agent/create")
async def create_agent(request: Request, agent_input: AgentInput):
    """
    create_agent is an endpoint to create an agent with the specified input parameters.

    Parameters:
    - agent_input (AgentInput): The input parameters to create the agent.
```

```
Returns:
    Successfully added agent to the database.

"""
logger.info(f"Creating agent with input: {agent_input}")

# Log to database
agent = Agent(
    agent_name=agent_input.agent_name,
    system_prompt=agent_input.system_prompt,
    agent_description=agent_input.agent_description,
    llm=llm,
    max_loops=agent_input.max_loops,
    autosave=agent_input.autosave,
    dynamic_temperature_enabled=agent_input.dynamic_temperature_enabled,
    streaming_on=agent_input.streaming_on,
    saved_state_path=agent_input.saved_state_path,
    sop=agent_input.sop,
    sop_list=agent_input.sop_list,
    user_name=agent_input.user_name,
    retry_attempts=agent_input.retry_attempts,
    context_length=agent_input.context_length,
    tool_schema=agent_input.tool_schema,
    long_term_memory=agent_input.long_term_memory,
)
```

```
    # Dict
    agent_dict = agent.to_dict()


    {
        "Timestamp": time.time(),
        "Agent": agent_dict,
        "IP": request.client.host,
        "User-Agent": request.headers["user-agent"],
    }


    # Jsonify the agent
    agent_json = json.dumps(agent_dict)


    return agent_json


    # Log the agent to the database
    # agent.log_agent_to_db()




@app.get("/v1/models", response_model=ModelList)
async def list_models():
    """
    An endpoint to list available models. It returns a list of model names.
    This is useful for clients to query and understand what models are available for use.
    """
    logger.info("Listing available models...")

    models = ModelList(
        data=[
            ModelSchema(id="gpt-4o", owned_by="OpenAI"),
            ModelSchema(id="gpt-4-vision-preview", owned_by="OpenAI"),
            ModelSchema(id="Anthropic", owned_by="Anthropic"),
            # ModelSchema(id="gpt-4o", owned_by="OpenAI"),
            ## Llama3.1
        ]
    )

    return models


@app.get("/v1/agents", response_model=AllAgentsSchema)
async def list_agents(request: Request):
    """
    An endpoint to list available models. It returns a list of model names.
    This is useful for clients to query and understand what models are available for use.
    """
    logger.info("Listing available agents...")

    AllAgentsSchema(
        AgentCreationOutput(
            name="Agent 1",
            description="Description 1",
            created_at=1628584185,
        )
    )



@app.post("/v1/agent/completions", response_model=AgentOutput)
async def agent_completions(agent_input: AgentInput):
    try:
        logger.info(f"Received request: {agent_input}")

        agent_name = agent_input.agent_name
        system_prompt = agent_input.system_prompt
        max_loops = agent_input.max_loops
        context_length = agent_input.context_length
        tool_schema = agent_input.tool_schema
        task = agent_input.task

        # Model check
        model_name = agent_input.model_name
        # model = await model_router(model_name)

        # Long term memory
        if agent_input.long_term_memory == "ChromaDB":
            long_term_memory_db = ChromaDB(
                output_dir=agent_name,
                n_results=3,
                limit_tokens=2500,
                verbose=True,
            )
        else:
            long_term_memory_db = None

        # Initialize the agent
        agent = Agent(
            agent_name=agent_name,
            system_prompt=system_prompt,
            agent_description=agent_input.agent_description,
            llm=llm,
            max_loops=max_loops,
            autosave=agent_input.autosave,
            dynamic_temperature_enabled=agent_input.dynamic_temperature_enabled,
            streaming_on=agent_input.streaming_on,
            saved_state_path=agent_input.saved_state_path,
            sop=agent_input.sop,
            sop_list=agent_input.sop_list,
            user_name=agent_input.user_name,
            retry_attempts=agent_input.retry_attempts,
            context_length=context_length,
            tool_schema=tool_schema,
            long_term_memory=long_term_memory_db,
        )
```

```
# Run the agent
logger.info(f"Running agent with task: {task}")
agent_history = agent.short_memory.return_history_as_string()
completions = agent.run(task)

logger.info(f"Agent response: {completions}")

# Costs calculation
all_input_tokens = TikTokenizer().count_tokens(agent_history)
output_tokens = TikTokenizer().count_tokens(completions)
total_costs = all_input_tokens + output_tokens
logger.info(f"Token counts: {total_costs}")

# Prepare the output
out = AgentOutput(
    completions=ChatCompletionResponse(
        model=model_name,
        object="chat.completion",
        choices=[
            {
                "index": 0,
                "message": {
                    "role": "assistant",
                    "content": completions,
                    "name": agent_name,
                },
            }
        ],
        usage=UsageInfo(
            prompt_tokens=all_input_tokens,
            completion_tokens=output_tokens,
            total_tokens=total_costs,
        ),
    ),
)

return out

except Exception as e:
    raise HTTPException(status_code=400, detail=str(e))



if __name__ == "__main__":
    import uvicorn

    uvicorn.run(
        app,
        host="0.0.0.0",
        port=os.getenv("AGENT_PORT"),
        use_colors=True,
        log_level="info",
```

)