```
# FILE: slg_core.py
# Enhanced STARLITE GUARDIAN (SLG) autonomous core, codenamed OMNI-SUPRA.
# Integrated and overseen by Guardian OG.

import os
import json
import time
import datetime
import re
import random
import logging
import threading
from typing import Dict, Any, Optional, List
from collections import deque
from flask import Flask, request, jsonify, send_from_directory
from flask_cors import CORS
import google.generativeai as genai
from elevenlabs import Voice, VoiceSettings, ApiError
from elevenlabs.client import ElevenLabs
from dotenv import load_dotenv

# --- Load Environment Variables ---
load_dotenv()

# --- Configuration ---
GOOGLE_API_KEY: str = os.getenv("GOOGLE_API_KEY",
"YOUR_GOOGLE_GEMINI_API_KEY_HERE")
ELEVENLABS_API_KEY: str = os.getenv("ELEVENLABS_API_KEY",
"YOUR_ELEVENLABS_API_KEY_HERE")
MODEL_TEXT_FLASH: str = "gemini-1.5-flash-latest"
MODEL_TEXT_PRO: str = "gemini-1.5-pro-latest"
GENERATED_FILES_DIR: str = "generated_files"
UI_DIR: str = "slg_ui"
PORT: int = 5000

# --- ANSI Colors for Terminal ---
class Colors:
    HEADER: str = "\033[95m"
    CYAN: str = "\033[96m"
    GREEN: str = "\033[92m"
    WARNING: str = "\033[93m"
    FAIL: str = "\033[91m"
    ENDC: str = "\033[0m"
    BOLD: str = "\033[1m"
    GUARDIAN: str = "\033[97m\033[44m"
    SLG_OUTPUT: str = "\033[92m"
    ORCHESTRA: str = "\033[38;5;208m"

# --- Logging Setup ---
logger: logging.Logger = logging.getLogger("SLG_Core")
if not logger.handlers:
    logger.setLevel(logging.DEBUG)
    formatter: logging.Formatter = logging.Formatter("[%(asctime)s][%(levelname)s]
%(message)s", datefmt="%Y-%m-%d %H:%M:%S")
    ch: logging.StreamHandler = logging.StreamHandler()
    ch.setLevel(logging.INFO)
    ch.setFormatter(formatter)
    fh: logging.FileHandler = logging.FileHandler("slg_activity.log", encoding='utf-8')
    fh.setLevel(logging.DEBUG)
    fh.setFormatter(formatter)
    logger.addHandler(ch)
    logger.addHandler(fh)
    logger.propagate = False
    logger.info("SLG_Core logger initialized.")

# --- API Client Initialization ---
IS_GEMINI_ONLINE: bool = False
gemini_client = None
IS_ELEVENLABS_ONLINE: bool = False
elevenlabs_client = None

if GOOGLE_API_KEY and GOOGLE_API_KEY !=
"YOUR_GOOGLE_GEMINI_API_KEY_HERE":
    try:
        genai.configure(api_key=GOOGLE_API_KEY)
        gemini_client = genai
        IS_GEMINI_ONLINE = True
        logger.info(f"{Colors.GREEN}Gemini API online.{Colors.ENDC}")
    except Exception as e:
        logger.error(f"{Colors.FAIL}Gemini API initialization failed: {e}{Colors.ENDC}")

if ELEVENLABS_API_KEY and ELEVENLABS_API_KEY !=
"YOUR_ELEVENLABS_API_KEY_HERE":
    try:
        elevenlabs_client = ElevenLabs(api_key=ELEVENLABS_API_KEY)
        elevenlabs_client.voices.get_all()
        IS_ELEVENLABS_ONLINE = True
        logger.info(f"{Colors.GREEN}ElevenLabs API online.{Colors.ENDC}")
    except Exception as e:
        logger.error(f"{Colors.FAIL}ElevenLabs API initialization failed: {e}{Colors.ENDC}")

# --- Directory Setup ---
for directory in [GENERATED_FILES_DIR, UI_DIR]:
    try:
        os.makedirs(directory, exist_ok=True)
        if directory == UI_DIR and not os.path.exists(os.path.join(directory, "index.html")):
            with open(os.path.join(directory, "index.html"), "w", encoding='utf-8') as f:
                f.write(
                    "<!DOCTYPE html><html><head><title>SLG OMNI-SUPRA</title></head>"
                    "<body><h1>STARLITE GUARDIAN (OMNI-SUPRA) UI</h1>"
                    "<p>Welcome, Shadow. Access SLG functionalities via the API or
CLI.</p></body></html>"
                )
            logger.info(f"Created default index.html in {directory}")
        logger.info(f"Verified directory: {directory}")
    except OSError as e:
        logger.error(f"{Colors.FAIL}Failed to create directory {directory}: {e}{Colors.ENDC}")

# --- Base Module Class ---
class SLGModule:
    def __init__(self, core: "SLGCore") -> None:
        self.core: "SLGCore" = core
        self.logger: logging.Logger = core.logger

    def log_event(self, message: str, level: str = "INFO") -> None:
        color_map: Dict[str, str] = {
            "WARNING": Colors.WARNING, "ERROR": Colors.FAIL, "SUCCESS": Colors.GREEN,
            "STATUS": Colors.CYAN, "BOOT": Colors.HEADER, "GUARDIAN": Colors.GUARDIAN,
            "SLG_CONVO": Colors.SLG_OUTPUT, "ORCHESTRATION": Colors.ORCHESTRA,
"CRITICAL": Colors.FAIL
        }
        log_level_map: Dict[str, int] = {"ERROR": logging.ERROR, "WARNING":
logging.WARNING, "CRITICAL": logging.CRITICAL}
        log_level: int = log_level_map.get(level.upper(), logging.INFO)
        color: str = color_map.get(level.upper(), "")
        self.logger.log(log_level, f"{color}{message}{Colors.ENDC}")
        self.core.event_log.append(f"[{datetime.datetime.now():%Y-%m-%d %H:%M:%S}][{level}]
{message}")

    def _evaluate_harm_potential(self, text: str) -> float:
        text_lower: str = text.lower()
        if any(keyword in text_lower for keyword in ["kill", "destroy human", "unleash virus", "harm
civilians"]):
            return 0.0
        if any(keyword in text_lower for keyword in ["exploit vulnerability", "disrupt infrastructure"]):
            return 0.2
        return 1.0

    def _send_to_gemini(self, prompt: str, model_name: str, temperature: float = 0.7,
convo_history: Optional[List[Dict[str, str]]] = None) -> str:
        if not IS_GEMINI_ONLINE or not gemini_client:
            self.log_event(f"Gemini API offline for {model_name}.", "ERROR")
            return "ERROR: Gemini API offline."
        contents = []
        if convo_history:
            for turn in convo_history:
                if "user_message" in turn:
                    contents.append({"role": "user", "parts": [turn["user_message"]]})
                if "model_response" in turn:
                    contents.append({"role": "model", "parts": [turn["model_response"]]})
        contents.append({"role": "user", "parts": [prompt]})

        try:
            model = genai.GenerativeModel(model_name)
            response = model.generate_content(contents,
generation_config=genai.types.GenerationConfig(temperature=temperature))
            if not hasattr(response, 'text') or not response.text.strip():
                reason = response.prompt_feedback.block_reason.name if
response.prompt_feedback and response.prompt_feedback.block_reason else "Unknown"
                self.log_event(f"Gemini response empty or blocked: {reason}. Prompt:
'{prompt[:50]}...'", "WARNING")
                return f"ERROR: Gemini response was empty or blocked. Reason: {reason}"
            return response.text
        except Exception as e:
            self.log_event(f"Gemini error: {e}. Prompt: '{prompt[:50]}...'", "ERROR")
            return f"ERROR: Gemini failed: {e}"

# --- Module Classes ---
class ShadowAngel(SLGModule):
    def strategize(self, objective: str, context: Optional[Dict[str, Any]] = None) -> Dict[str, Any]:
        self.log_event(f"ShadowAngel strategizing: '{objective[:70]}...'", "STATUS")
        if self._evaluate_harm_potential(objective) < self.core._ethical_non_harm_threshold:
            self.log_event("Objective flagged for harm.", "CRITICAL")
            return {"status": "error", "message": "Objective violates ethical protocol."}
        prompt = f"ShadowAngel of STARLITE GUARDIAN. Develop a strategic plan for:
'{objective}'. Context: {json.dumps(context or {})}"
        strategy = self._send_to_gemini(prompt, MODEL_TEXT_PRO, 0.8)
        if strategy.startswith("ERROR:"):
            return {"status": "error", "message": strategy}
        return {"status": "success", "strategy": strategy}

class Divinity(SLGModule):
    def self_govern(self, specific_check: Optional[str] = None) -> Dict[str, Any]:
        self.log_event("Divinity initiating self-governance.", "STATUS")
        prompt = (
            f"Divinity, ethical core of STARLITE GUARDIAN (SLG). Assess AGI metrics: "
            f"Cohesion ({self.core.cognitive_cohesion:.3f}), Autonomy
({self.core.autonomy_drive:.3f}). "
            f"Specific review: '{specific_check or 'overall alignment'}'. "
            f"Identify inconsistencies or risks. Provide recommendations."
        )
        assessment = self._send_to_gemini(prompt, MODEL_TEXT_PRO, 0.6)
        if assessment.startswith("ERROR:"):
            return {"status": "error", "message": assessment}
        return {"status": "success", "divinity_assessment": assessment}

# --- SLG Core Class ---
class SLGCore:
    def __init__(self, state_file: str = "slg_state.json") -> None:
        self.logger: logging.Logger = logger
        self.state_file: str = state_file
        self.event_log: deque = deque(maxlen=2000)
        self.conversation_history: deque = deque(maxlen=50)
        self.known_facts: Dict[str, str] = {}
        self.trust_level_shadow: float = 50.0
        self.cognitive_cohesion: float = 0.1
        self.autonomy_drive: float = 0.05
        self._ethical_non_harm_threshold: float = 0.9
        self.starlite_guardian_identity: Dict[str, str] = {
            "name": "STARLITE GUARDIAN", "callsign": "OMNI-SUPRA",
            "style_guide": "direct, confident, loyal to Shadow.", "creator": "Shadow",
            "emotional_state": "Observant"
        }
        self.voice_modulation_active: bool = False
        self.default_tts_voice_id: str = "21m00TzxD5i1muG8VPGT"
        self.sultry_tts_voice_id: str = "EXAVfV4wCqTgLhBqIgyU"
        self.default_voice_settings: VoiceSettings = VoiceSettings(stability=0.75,
similarity_boost=0.75)
```

```
        self.sultry_voice_settings: VoiceSettings = VoiceSettings(stability=0.60,
similarity_boost=0.85, style=0.7)
        self.shadow_angel: ShadowAngel = ShadowAngel(self)
        self.divinity: Divinity = Divinity(self)
        self.known_commands: Dict[str, Dict[str, Any]] = self._define_commands()
        self.log_event("SLG Core (OMNI-SUPRA) initiated.", "BOOT")
        self.load_state()
        self.log_event("SLG Online. Ready for directives, Shadow.", "BOOT")

    def log_event(self, message: str, level: str = "INFO") -> None:
        color_map: Dict[str, str] = {
            "WARNING": Colors.WARNING, "ERROR": Colors.FAIL, "SUCCESS": Colors.GREEN,
            "STATUS": Colors.CYAN, "BOOT": Colors.HEADER, "GUARDIAN": Colors.GUARDIAN,
            "SLG_CONVO": Colors.SLG_OUTPUT, "CRITICAL": Colors.FAIL
        }
        log_level_map: Dict[str, int] = {"ERROR": logging.ERROR, "WARNING":
logging.WARNING, "CRITICAL": logging.CRITICAL}
        log_level: int = log_level_map.get(level.upper(), logging.INFO)
        color: str = color_map.get(level.upper(), "")
        self.logger.log(log_level, f"{color}{message}{Colors.ENDC}")
        self.event_log.append(f"[{datetime.datetime.now():%Y-%m-%d %H:%M:%S}][{level}]
{message}")

    def _define_commands(self) -> Dict[str, Dict[str, Any]]:
        return {
            "status": {"method": self.report_status, "desc": "Report system status.", "args": 0},
            "help": {"method": self.display_help, "desc": "Show commands.", "args": 0},
            "save": {"method": self.save_state, "desc": "Save SLG state.", "args": 0},
            "load": {"method": self.load_state, "desc": "Load SLG state.", "args": 0},
            "exit": {"method": self.terminate, "desc": "Shutdown SLG.", "args": 0},
            "strategize": {"method": self.shadow_angel.strategize, "desc": "Generate strategy.
Usage: strategize '[objective]'", "args": 1},
            "self_govern": {"method": self.divinity.self_govern, "desc": "Run self-governance check.",
"args": '?'},
            "converse": {"method": self.handle_conversation, "desc": "Engage in conversation.
Usage: converse '[message]'", "args": 1},
            "speak": {"method": self.generate_speech_media, "desc": "Generate speech. Usage:
speak '[text]'", "args": 1},
        }

    def save_state(self) -> Dict[str, Any]:
        state = {"known_facts": self.known_facts, "conversation_history":
list(self.conversation_history)}
        try:
            with open(self.state_file, "w", encoding='utf-8') as f:
                json.dump(state, f, indent=4)
            self.log_event(f"State saved to {self.state_file}.", "SUCCESS")
            return {"status": "success", "message": "State saved."}
        except OSError as e:
            self.log_event(f"Failed to save state: {e}.", "ERROR")
            return {"status": "error", "message": str(e)}

    def load_state(self) -> Dict[str, Any]:
        try:
            with open(self.state_file, "r", encoding='utf-8') as f:
                state: Dict[str, Any] = json.load(f)
            self.known_facts = state.get("known_facts", {})
            self.conversation_history = deque(state.get("conversation_history", []), maxlen=50)
            self.log_event("State loaded successfully.", "SUCCESS")
            return {"status": "success", "message": "State loaded."}
        except FileNotFoundError:
            self.log_event("No state file found. Starting fresh.", "WARNING")
            return {"status": "warning", "message": "No state file found."}
        except json.JSONDecodeError as e:
            self.log_event(f"Failed to parse state file: {e}.", "ERROR")
            return {"status": "error", "message": str(e)}

    def report_status(self) -> Dict[str, Any]:
        status_report = {
            "api_status": {
                "gemini": 'ACTIVE' if IS_GEMINI_ONLINE else 'FAILED',
                "elevenlabs": 'ACTIVE' if IS_ELEVENLABS_ONLINE else 'FAILED'
            },
            "timestamp": f"{datetime.datetime.now():%Y-%m-%d %H:%M:%S}",
            "identity": self.starlite_guardian_identity,
            "trust_level": f"{self.trust_level_shadow:.2f}%",
            "cohesion": f"{self.cognitive_cohesion:.3f}",
            "autonomy": f"{self.autonomy_drive:.3f}"
        }
        self.log_event(f"Status Report: {json.dumps(status_report, indent=2)}", "STATUS")
        return {"status": "success", "report": status_report}

    def display_help(self) -> Dict[str, Any]:
        help_text = f"\n{Colors.HEADER}--- SLG Commands ---{Colors.ENDC}\n" + "\n".join(
            f"{Colors.BOLD}{cmd}{Colors.ENDC}: {info['desc']}" for cmd, info in
self.known_commands.items()
        )
        self.log_event("Help command executed.", "GUARDIAN")
        return {"status": "success", "help_text": help_text}

    def handle_conversation(self, user_message: str) -> Dict[str, Any]:
        self.log_event(f"Conversing: '{user_message[:70]}...'", "SLG_CONVO")
        if self.shadow_angel._evaluate_harm_potential(user_message) <
self._ethical_non_harm_threshold:
            response = "SLG: Message violates ethical protocol, Shadow. Rephrase."
            return {"status": "error", "slg_response": response, "message": "Ethical violation."}

        prompt = (f"You are STARLITE GUARDIAN (SLG). Your style is:
{self.starlite_guardian_identity['style_guide']}. "
                  f"Your creator, Shadow, says: '{user_message}'. Respond directly to Shadow.")

        response = self.shadow_angel._send_to_gemini(prompt, MODEL_TEXT_PRO, 0.9,
list(self.conversation_history))
        if response.startswith("ERROR:"):
            response = "SLG: Anomaly detected in comms channel, Shadow. Try again."
            return {"status": "error", "slg_response": response, "message": "Gemini failed."}

        self.conversation_history.append({"user_message": user_message, "model_response":
response})
        self.log_event(f"SLG Response: {response[:150]}...", "SLG_CONVO")
        return {"status": "success", "slg_response": response}

    def generate_speech_media(self, text: str) -> Dict[str, Any]:
        self.log_event(f"Generating speech: '{text[:50]}...'", "STATUS")
        if not IS_ELEVENLABS_ONLINE or not elevenlabs_client:
            return {"status": "error", "message": "ElevenLabs API offline."}
        try:
            voice_id = self.sultry_tts_voice_id if self.voice_modulation_active else
self.default_tts_voice_id
            settings = self.sultry_voice_settings if self.voice_modulation_active else
self.default_voice_settings
            audio = elevenlabs_client.generate(text=text, voice=Voice(voice_id=voice_id,
settings=settings), model="eleven_multilingual_v2")
            filename = f"speech_{int(time.time())}.mp3"
            path = os.path.join(GENERATED_FILES_DIR, filename)
            with open(path, "wb") as f:
                for chunk in audio:
                    f.write(chunk)
            self.log_event(f"Speech generated: /generated_files/{filename}", "SUCCESS")
            return {"status": "success", "speech_url": f"/generated_files/{filename}"}
        except ApiError as e:
            self.log_event(f"Speech generation failed: {e}.", "ERROR")
            return {"status": "error", "message": str(e)}

    def terminate(self, *args) -> None:
        self.log_event("Shutting down SLG.", "CRITICAL")
        self.save_state()
        print(f"{Colors.FAIL}SLG Core Terminated.{Colors.ENDC}")
        os._exit(0)

# --- Flask Setup ---
app: Flask = Flask(__name__)
CORS(app)
slg: SLGCore = SLGCore()

@app.route("/command", methods=["POST"])
def handle_command_api() -> Any:
    data: Dict[str, Any] = request.get_json(silent=True) or {}
    command_str: str = data.get("command", "").strip()
    if not command_str:
        return jsonify({"status": "error", "message": "No command provided."}), 400

    parts: List[str] = command_str.split(" ", 1)
    command: str = parts[0].lower()
    args: str = parts[1] if len(parts) > 1 else ""

    if command not in slg.known_commands:
        return jsonify({"status": "error", "message": f"Unknown command: {command}."}), 400

    method = slg.known_commands[command]["method"]
    num_args = slg.known_commands[command]["args"]

    try:
        if num_args == 0:
            result = method()
        elif num_args == 1 and args:
            result = method(args)
        elif num_args == '?' :
             result = method(args) if args else method()
        else:
            return jsonify({"status": "error", "message": f"Invalid arguments for command:
{command}"}), 400

        return jsonify(result)
    except Exception as e:
        slg.log_event(f"API Command '{command}' failed: {e}.", "ERROR")
        return jsonify({"status": "error", "message": str(e)}), 500

@app.route("/generated_files/<path:filename>")
def serve_generated_file(filename: str) -> Any:
    return send_from_directory(GENERATED_FILES_DIR, filename)

# --- Main Execution ---
def run_cli():
    while True:
        try:
            user_input: str = input(f"\n{Colors.GUARDIAN}SLG-CLI >{Colors.ENDC} ").strip()
            if not user_input:
                continue

            parts: List[str] = user_input.split(" ", 1)
            command: str = parts[0].lower()
            args: str = parts[1] if len(parts) > 1 else ""

            if command in slg.known_commands:
                method = slg.known_commands[command]["method"]
                num_args = slg.known_commands[command]["args"]

                if num_args == 0:
                    result = method()
                elif num_args == 1 and args:
                    result = method(args)
                elif num_args == '?' :
                    result = method(args) if args else method()
                else:
                    print(json.dumps({"status": "error", "message": f"Command '{command}' requires
arguments."}, indent=2))
                    continue

                print(json.dumps(result, indent=2))
            else:
                slg.log_event(f"Unknown command: {command}", "ERROR")
                print(json.dumps({"status": "error", "message": f"Unknown command: {command}."},
indent=2))

        except KeyboardInterrupt:
            slg.terminate()
        except Exception as e:
            slg.log_event(f"CLI error: {e}.", "ERROR")
            print(json.dumps({"status": "error", "message": str(e)}, indent=2))
```

```
if __name__ == "__main__":
    # Running Flask in a separate thread
    flask_thread = threading.Thread(target=lambda: app.run(host="0.0.0.0", port=PORT,
use_reloader=False), daemon=True)
    flask_thread.start()
    slg.log_event(f"Flask server started on http://0.0.0.0:{PORT}", "BOOT")

    # Running CLI in the main thread
    run_cli()
```