ChatGPT

AI Assistant GUI Prototype (Streamlit)

This document outlines a modular Streamlit-based **AI Assistant GUI** with a retro minimalist interface. It includes a conversational chat UI, optional visual insight panels (2D/3D), memory persistence, and a development status tracker. The structure is organized into separate files for clarity and maintainability.

Directory Structure

- **app.py** – Main Streamlit app, managing UI layout, chat interface, and control flow.
- **memory.py** – Memory management (short-term chat history in session, and persistent storage in JSON).
- **visualizer.py** – Visualization utilities for 2D/3D insights using Plotly, Matplotlib, and PyThreeJS (via HTML).
- **rag_interface.py** – Retrieval-Augmented Generation interface (stub for integrating knowledge base or LangChain retrieval).
- **response_generator.py** – AI response generation logic (could integrate with LangChain or OpenAI API).
- **style.css** – Custom CSS for retro, pixel-inspired styling (monospace font, dark theme).
- **status.json** – JSON file tracking development status (module completeness, test coverage, deployment readiness).

File: app.py

This Streamlit app provides a password-gated chat interface with optional visuals and data preview panels. It uses Streamlit's chat elements (`st.chat_message`, `st.chat_input`) to create a conversational UI [1]. A sidebar offers toggles to display visualizations and data previews. User authentication is implemented via a simple password prompt at launch [2]. The app also reads a `status.json` to display development status if needed.

```python
import streamlit as st
from streamlit import session_state as state
import json
import os
from pathlib import Path

import memory, visualizer, rag_interface, response_generator

# --- Basic config ---
st.set_page_config(page_title="AI Assistant", layout="wide")
# Load custom CSS for retro style
css_file = Path("style.css")
if css_file.exists():
    st.markdown(f"<style>{css_file.read_text()}</style>",
unsafe_allow_html=True)

# --- Authentication (password gate) ---
PASSWORD = st.secrets["APP_PASSWORD"] if "APP_PASSWORD" in st.secrets else
os.getenv("APP_PASSWORD", "admin")
if not state.get("authenticated"):
    pwd = st.text_input("Enter password:", type="password")
    if pwd == PASSWORD:
        state.authenticated = True
        st.experimental_rerun()
    else:
        st.stop()  # Prevent access until correct password is provided
# (Using a simple password check via st.text_input for basic security [2] )

# --- Sidebar controls ---
st.sidebar.title("Settings")
show_visual = st.sidebar.checkbox("Visualize Responses", value=False)
show_data   = st.sidebar.checkbox("Show Data Preview", value=False)
show_status = st.sidebar.checkbox("Show Dev Status", value=False)

# --- Initialize session state for messages ---
if "messages" not in state:
    state.messages = []
    # Attempt to load persistent chat history (if exists)
    loaded_history = memory.load_history()
    if loaded_history:
        state.messages = loaded_history

# --- Greeting (Assistant's welcome message) ---
if len(state.messages) == 0:
    welcome_text = "Hello! I'm your AI assistant. How can I help you today?"
    with st.chat_message("assistant"):
        st.markdown(welcome_text)
    state.messages.append({"role": "assistant", "content": welcome_text})

st.title("  AI Assistant")
st.markdown("*(Ask me anything!)*")

# --- Display chat history ---
for msg in state.messages:
    with st.chat_message(msg["role"]):
        st.markdown(msg["content"])

# --- User input ---
if user_input := st.chat_input("Type your message:"):  # sticky chat input at
page bottom
    # Display user query in chat
    with st.chat_message("user"):
        st.markdown(user_input)
    state.messages.append({"role": "user", "content": user_input})

    # Generate response (with optional retrieval context)
    with st.chat_message("assistant"):
        with st.spinner("Thinking..."):
            # Use RAG interface to get context docs (if any)
            context = rag_interface.get_relevant_docs(user_input)
            response_text = response_generator.generate_response(user_input,
context=context)
            # Display assistant response text
            st.markdown(response_text)
            # If visual output is enabled, include generated visuals
            if show_visual:
                # Example: display Plotly, Matplotlib, and 3D visualizations
                tabs = st.tabs(["Plotly Chart", "Matplotlib", "3D View"])
                with tabs[0]:
                    fig = visualizer.get_plotly_fig()
                    st.plotly_chart(fig, use_container_width=True)
                with tabs[1]:
                    fig2 = visualizer.get_matplotlib_fig()
                    st.pyplot(fig2)
                with tabs[2]:
                    html_3d = visualizer.get_3d_html()
                    st.components.v1.html(html_3d, height=400)
            if show_data:
                # Show a preview of persistent memory or retrieved data
                st.divider()
                st.subheader("Data Preview")
                data_preview = {"chat_history": state.messages}  # example:
current chat memory
                st.json(data_preview)
    # Save assistant response in chat history
    state.messages.append({"role": "assistant", "content": response_text})
    # Persist the updated conversation to file
    memory.save_history(state.messages)

# --- Developer Status (optional, in sidebar) ---
if show_status:
    st.sidebar.subheader("Development Status")
    try:
        status_data = memory.load_status()
        st.sidebar.json(status_data)
    except FileNotFoundError:
        st.sidebar.write("_status.json not found_")
```

File: memory.py

The `memory` module handles short-term and long-term memory. Short-term memory (conversation history) lives in `st.session_state` for each user session [3]. Persistent memory is stored in JSON files on disk (here in a `data/` folder) to retain information across sessions, analogous to LangChain's file-based chat history mechanism [4]. This module provides functions to load/save chat history and to load the development status file.

```python
import json
from pathlib import Path

# Define file paths (ensure 'data' directory exists)
DATA_DIR = Path("data")
DATA_DIR.mkdir(exist_ok=True)
HISTORY_FILE = DATA_DIR / "chat_history.json"
STATUS_FILE  = Path("status.json")

def load_history():
    """Load persisted chat history from JSON, or return empty list."""
    if HISTORY_FILE.exists():
        try:
            with open(HISTORY_FILE, 'r') as f:
                return json.load(f)
        except json.JSONDecodeError:
            return []
    return []

def save_history(messages):
    """Save chat history (list of message dicts) to JSON file."""
    with open(HISTORY_FILE, 'w') as f:
        json.dump(messages, f, indent=2)

def load_status():
    """Load development status info from status.json."""
    if STATUS_FILE.exists():
        with open(STATUS_FILE, 'r') as f:
            return json.load(f)
    else:
        raise FileNotFoundError("Status file not found")
```

File: visualizer.py

The `visualizer` module produces visual insight objects. It includes examples for Plotly (2D chart), Matplotlib (2D plot), and a placeholder for PyThreeJS/3D rendering. In a real app, these functions would generate charts based on data or the conversation context. Here we show simple static visuals for demonstration. (Interactive 3D can be integrated by exporting a scene to HTML and embedding it, e.g. using PyVista or Panel to wrap Three.js [5] .)

```
import plotly.graph_objs as go
import matplotlib.pyplot as plt
from io import BytesIO

def get_plotly_fig():
    """Return a sample Plotly figure (e.g., a sine wave scatter)."""
    import numpy as np
    x = np.linspace(0, 2*np.pi, 100)
    y = np.sin(x)
    fig = go.Figure(data=go.Scatter(x=x, y=y, mode='lines', name='sin'))
    fig.update_layout(title="Sine Wave", template="plotly_dark",
                      margin=dict(l=20, r=20, t=40, b=20))
    return fig

def get_matplotlib_fig():
    """Return a sample Matplotlib figure (e.g., cosine curve)."""
    import numpy as np
    x = np.linspace(0, 2*np.pi, 100)
    y = np.cos(x)
    plt.figure(figsize=(4,3))
    plt.plot(x, y, color='orange')
    plt.title("Cosine Wave")
    plt.tight_layout()
    fig = plt.gcf()  # get current figure
    return fig

def get_3d_html():
    """Return HTML string for a placeholder 3D visualization."""
    # TODO: Replace with real 3D rendering (e.g., PyThreeJS via PyVista export)
 [5] .
    html_content = """
    <div style='text-align:center; color: cyan; font-family: monospace;'>
        [3D Visualization Placeholder: Imagine a rotating 3D object here]
    </div>
    """
    return html_content
```

File: rag_interface.py

This module is a stub for Retrieval-Augmented Generation (RAG) capabilities. In a complete system, it would interface with a knowledge base or vector store to fetch relevant context documents for a query, which the AI model can then use to formulate a response [6] . For now, it returns an empty context or a dummy result.

```
# rag_interface.py

def get_relevant_docs(user_query):
    """
    Retrieve relevant documents or data for the given query (stub).
    In practice, this could query a vector database or use LangChain retrieval.
    """
    # Example: return [] or dummy data
    return []
```

(RAG systems combine a retrieval step with generation – e.g. searching a corpus and providing results to the LLM to ground its answer [6] *. This stub can be extended with LangChain's retrieval QA chains or similar.)*

File: response_generator.py

The `response_generator` module handles the AI's response generation. This is where integration with an LLM (like OpenAI GPT via LangChain or direct API calls) would occur. It could take the user prompt and optional context (from `rag_interface`) to produce a reply. In this prototype, we provide a simple placeholder implementation. (One could plug in a LangChain LLMChain or ConversationChain here.)

```
# response_generator.py

def generate_response(user_input, context=None):
    """
    Generate the assistant's response to the user_input.
    `context` can be used to include retrieved info (if any) in the prompt.
    """
    # Placeholder logic for demo purposes:
    if context:

# If context documents are provided, you might concatenate them into the prompt.
        knowledge = " ".join([doc for doc in context])
    else:
        knowledge = ""
    # For now, just echo the user input with a simple transformation
    response = f" : You said '{user_input}'. (This is a placeholder response.)"
    return response

# (In a real implementation, this function could call an LLM API or LangChain
pipeline
#  to generate a context-aware answer, possibly using prompt templates, etc.)
```

File: style.css

This CSS file defines the retro, pixel-art inspired style for the app. It uses a monospace font for a terminal-like feel, high-contrast colors (dark background with bright text), and basic styling for Streamlit elements. The design is minimalist to ensure usability. *(Note: Streamlit's class names may change; this CSS targets common elements for demonstration.)*

```css
/* Retro minimalist theme */
body {
  background-color: #111;  /* dark background */
  color: #33ff33;          /* neon green text */
  font-family: "Courier New", monospace; /* monospace font for retro feel */
}
[data-testid="stSidebar"] {
  background-color: #222;
  color: #33ff33;
}
h1, h2, h3, .stMarkdown, .stChatMessage, .stButton, .stTextInput {
  font-family: "Courier New", monospace;
  color: #33ff33;
}
.stChatMessage .stMarkdown p {
  /* Chat message text styling */
  color: #33ff33;
}
.stButton>button, .stTextInput>div>input {
  background-color: #333;
  color: #33ff33;
  border: 1px solid #33ff33;
}
```

File: status.json

This JSON file tracks the development progress of each module, testing coverage, and deployment readiness. It can be manually or programmatically updated as development progresses. The app can load and display this for developers via the sidebar toggle.

```json
{
  "modules": {
    "app":        { "completed": true,  "tested": false },
    "memory":     { "completed": true,  "tested": false },
    "visualizer": { "completed": true,  "tested": false },
    "rag_interface": { "completed": true,  "tested": false },
    "response_generator": { "completed": true, "tested": false }
  },
  "test_coverage": 0,
  "deployment_ready": false
}
```

(Initially, all modules are written but not fully tested; test coverage and deployment readiness can be updated as the project matures.)

Testing and Multi-User Considerations

Testing Frameworks: For the Python backend logic (e.g., functions in `memory.py`, `response_generator.py`), **Pytest** is recommended to write unit tests and integration tests. Streamlit's new app testing API (`st.testing.AppTest`) allows simulating app runs and checking outputs, which can be integrated with pytest as well [7]. For the GUI itself, automated end-to-end tests can be implemented using **Playwright** or **Selenium**. Playwright is a modern choice with a pytest plugin for convenient usage [8]. It can launch the Streamlit app in headless mode and simulate user interactions (typing, clicking toggles, etc.), verifying that responses and UI updates occur as expected. There are examples of using Playwright to validate Streamlit components' output in real apps [9]. Selenium WebDriver is an alternative for browser-based testing if Playwright is not available, though it may require more setup.

Multi-User Sessions: By default, Streamlit handles each user session separately – each connected user gets their own `st.session_state` context (variables are unique per session) [3]. This design prevents session data from leaking between users. However, to fully support multi-user scenarios, the app should ensure any persistent storage is also separated per user. For example: - Maintain distinct JSON files or namespaces for each user's data (e.g., name files by a user ID or username). - If using a database or vector store for RAG, include session/user identifiers in queries to fetch only that user's allowed data. - Avoid using `st.cache_data` or `st.cache_resource` for user-specific information, as cached data might be shared across users if not scoped properly. Rely on session state and per-user file storage to keep data isolated.

To adapt this prototype for multiple users, one could implement a login system with unique user accounts (instead of a single password). Upon login, initialize separate `session_state` keys and file paths for that user. The `memory` module functions can take a user identifier to load/save different files (e.g., `data/history_<username>.json`). This ensures **secure data separation** between users' conversations and memory. Streamlit's session state combined with user-specific file management will keep each user's chat history and data private to their session.

References:

- Streamlit chat UI elements and session state usage [1] [10]
- Basic password gating via Streamlit input [2]
- Session state is per user-session (not shared between users) [3]
- Persistent chat history stored in file (similar to LangChain's FileChatMessageHistory) [4]
- 3D visualization via PyThreeJS/Panel HTML embedding [5]
- Retrieval-Augmented Generation (RAG) concept [6]
- Streamlit app testing with pytest (AppTest framework) [7]
- Playwright testing for Streamlit UIs (pytest plugin for browser automation) [8] and [9]

[1] [10] Build a basic LLM chat app - Streamlit Docs
https://docs.streamlit.io/develop/tutorials/chat-and-llm-apps/build-conversational-apps

[2] How to set login password for my streamlit app - Custom Components - Streamlit
https://discuss.streamlit.io/t/how-to-set-login-password-for-my-streamlit-app/13150

[3] Session State - Streamlit Docs
https://docs.streamlit.io/develop/api-reference/caching-and-state/st.session_state

[4] FileChatMessageHistory — LangChain documentation
https://api.python.langchain.com/en/latest/community/chat_message_histories/
langchain_community.chat_message_histories.file.FileChatMessageHistory.html

[5] stpyvista: Show PyVista 3D visualizations in Streamlit - Show the Community! - Streamlit
https://discuss.streamlit.io/t/stpyvista-show-pyvista-3d-visualizations-in-streamlit/31802

[6] Build an LLM RAG Chatbot With LangChain – Real Python
https://realpython.com/build-llm-rag-chatbot-with-langchain/

[7] App testing - Streamlit Docs
https://docs.streamlit.io/develop/api-reference/app-testing

[8] Test your Streamlit dashboard using Playwright · Stef Smeets
https://www.stefsmeets.nl/posts/streamlit-pytest/

[9] How to test a component - Custom Components - Streamlit
https://discuss.streamlit.io/t/how-to-test-a-component/62698