ChatGPT

Voice Agent EX3 Code Walkthrough

This document provides a line-by-line explanation of the files that make up the EX3 Voice Agent application. Each section describes what the code does, how data flows through the system, and how different components interact.

Overview

At a high level, this project implements a FastAPI server that supports a voice-driven chat loop. The browser client records the user's voice, sends audio to the server for transcription, invokes a language model to generate a reply, then synthesizes speech to return a spoken response. The code is modular; separate modules handle automatic speech recognition (ASR), language model (LLM), text-to-speech (TTS), and conversation state.

main.py

Purpose

`main.py` sets up the FastAPI web server and defines endpoints for the UI home page, resetting conversation history, transcribing audio, generating a reply, and synthesizing speech.

Walkthrough

1. **Imports**: The first lines import standard library modules for path handling, UUID generation, and temporary file management (`os`, `uuid`, `tempfile`, `shutil`) followed by FastAPI and its middleware, and Pydantic's `BaseModel` for request validation. It also imports custom modules from `src` (the ASR, LLM, TTS, and state modules).

2. **FastAPI app and CORS**: The `app` variable instantiates a FastAPI application with a title and version. Middleware is added to enable cross-origin requests from any origin, allowing the web client to interact with the API without restrictions.

3. **Conversation state**: A single `ConversationState` object is created. It reads `MAX_TURNS` from the environment (default `8`) and uses this to limit how many turns of conversation history are kept. This global state holds previous user and assistant messages for each session.

4. **Request model**: `TextIn` is a Pydantic model with a single `text: str` field; it validates JSON payloads for the `/reply` and `/speak` endpoints.

5. **Home page** (`GET "/"`): The `home` handler reads `client/index.html` into a string and returns it as an HTMLResponse. This serves the demo UI when users visit the root URL.

6. **Reset endpoint** (`POST "/reset"`): The `reset` handler clears the conversation history by calling `state.reset()` and returns `{ "ok": True }`. This allows the client to start a new conversation.

7. **Transcription endpoint** (`POST "/transcribe"`): The `asr_endpoint` accepts an `engine` form field (default "whisper") and a file upload. It writes the uploaded WebM audio to a temporary WAV file, calls `transcribe` from `src.asr` with the chosen engine and optional Google API key, then returns JSON containing the recognized text and a confidence estimate. The temporary directory is deleted afterwards.

8. **Reply endpoint** (`POST "/reply"`): The `reply_endpoint` receives JSON with a `text` field, appends the user's message to conversation state, calls `chat` from `src.llm` with the list of messages to get a reply, appends the assistant's reply to state, and returns `{"reply": reply}`.

9. **Speech synthesis endpoint** (`POST "/speak"`): The `speak_endpoint` accepts JSON with a `text` field, generates a temporary WAV file using `synthesize_to_wav` from `src.tts`, and returns the file via FastAPI's `FileResponse` so the browser can play the audio.

Overall, `main.py` orchestrates the interplay between the web client and the underlying ASR, LLM, and TTS components.

src/asr.py

Purpose

This module performs speech recognition. It supports two engines: Google SpeechRecognition (via the `speech_recognition` package) and OpenAI Whisper (via the `whisper` package).

Walkthrough

1. **Imports**: The file imports `os` and `io` (unused), `Tuple` from `typing`, and the `speech_recognition` package.

2. **Google transcription**: `transcribe_google` takes a WAV file path and optional API key. It creates a `Recognizer`, loads the audio from the file, and uses `recognize_google` to get the text. Since the API does not provide confidence scores via the SpeechRecognition wrapper, the function returns a constant 0.8 for confidence. Exceptions from `speech_recognition` are caught and converted to empty text and 0.0 confidence or raised as runtime errors.

3. **Whisper transcription**: `transcribe_whisper` imports the `whisper` library, loads a model (default "base"), transcribes the audio, retrieves the recognized text, and computes a proxy confidence as `1 - no_speech_prob` from the first segment. It returns the text and confidence.

4. **Unified interface**: `transcribe` is a wrapper that selects the engine based on the `engine` parameter. If `engine == "google"`, it calls `transcribe_google`, otherwise defaults to Whisper.

This modular design allows switching between local Whisper and cloud-based Google recognition via environment variables.

src/llm.py

Purpose

This module provides two back-ends for generating replies: Ollama (a local LLM server) and a fallback Hugging Face pipeline.

Walkthrough

1. **Imports**: It imports `os`, `json`, and `httpx` to make HTTP requests. It also imports `List` and `Dict` types.

2. **chat_ollama**: This function constructs a POST request to the Ollama chat endpoint. It reads `OLLAMA_URL` from environment (default `http://localhost:11434/api/chat`), builds a payload containing the model name and message list, and posts it using `httpx`. It returns the content of the `"message"` field from the response JSON. Errors will propagate via `r.raise_for_status()`.

3. **chat_hf**: This fallback uses Hugging Face's text generation pipeline. It concatenates previous messages into a prompt labelled "User:" and "Assistant:" lines. It calls the pipeline with a maximum of 128 new tokens and a temperature of 0.7, and returns the text after the last "Assistant:" label. The default model is read from `HF_MODEL`, or `TinyLlama` if not set.

4. **chat**: This top-level function reads the `LLM_BACKEND` environment variable. If it is `"hf"` (the default), it calls `chat_hf`; otherwise it calls `chat_ollama` with the specified model name.

The result is a simple abstraction that picks the appropriate generator based on environment variables.

src/tts.py

Purpose

This module synthesizes speech using the `pyttsx3` library.

Walkthrough

1. **Imports**: It imports `os` and `tempfile` (not used) plus `pyttsx3`.

2. **synthesize_to_wav**: This function initializes a pyttsx3 engine and optionally sets the speaking rate (`VOICE_RATE` environment variable) and voice by name (`VOICE_NAME`). It then calls `save_to_file` with the input text and output file path and runs the engine to generate audio. The function returns the output path.

pyttsx3 is a simple, offline TTS library; switching to other TTS systems can be done in future by modifying this function.

src/state.py

Purpose

This module maintains a ring buffer of conversation history so the LLM can reference previous turns up to a limit.

Walkthrough

1. **Imports and class definition**: It uses `collections.deque` to implement the ring buffer and defines a `ConversationState` class with `max_turns` and a default `system_prompt` instructing the assistant to be concise.

2. **reset**: Clears the history deque.

3. **add_user / add_assistant**: Appends a message dictionary containing the role (`"user"` or `"assistant"`) and content, then calls `_trim` to enforce the size limit.

4. **get_messages**: Returns a list combining the system prompt and all stored messages. This list is passed to the LLM.

5. **_trim**: While the history length exceeds `max_turns * 2` (each turn is a user–assistant pair), it removes the oldest message.

This ensures the LLM's context never grows beyond the configured number of turns.

client/index.html

Purpose

This HTML file implements a minimal user interface for recording voice, selecting the ASR engine, resetting conversation state, and displaying messages.

Walkthrough

1. **Structure and styles**: The document includes a title and basic CSS for chat bubbles and buttons.

2. **Controls**: It defines a "Record" button for audio input, a dropdown to choose between Whisper and Google for ASR, and a "Reset" button to clear the conversation. A `div#chat` displays messages and an `audio` element plays synthesized speech.

3. **Event handlers**: JavaScript functions handle recording with the MediaRecorder API, converting the WebM audio to WAV, posting it to `/transcribe`, updating the chat bubbles with recognized text

and the assistant's reply, requesting TTS from `/speak`, and playing the returned WAV file. It also resets conversation state when the reset button is clicked.

4. **WAV conversion**: Helper functions convert recorded WebM audio into a mono WAV format for the server.

This file shows how the client interacts with the API routes to create a conversational loop.

requirements.txt & Dependencies

The `requirements.txt` file lists third-party packages needed to run the project, including FastAPI and its dependencies, ASR packages (`SpeechRecognition`, `openai-whisper`), the `pydub` and `soundfile` libraries for audio I/O, `transformers` and `accelerate` for the HF LLM backend, and `pyttsx3` for speech synthesis. These packages must be installed in a virtual environment as shown in the README.

Execution Flow Summary

1. The user accesses the web client (index.html) which allows them to record audio and choose the ASR engine.
2. When audio is recorded, the browser converts it to WAV and sends it to the server's `/transcribe` endpoint. The server uses `src/asr.py` to transcribe speech into text.
3. The transcribed text is sent to the `/reply` endpoint. `main.py` appends this text to conversation state and invokes `src.llm.chat` to generate the assistant's reply.
4. The reply is returned to the browser and displayed. The browser also posts the reply text to `/speak`, triggering `src.tts.synthesize_to_wav` to produce a WAV file.
5. The WAV file is streamed back to the client and played. The user can then record another message, and the loop continues, maintaining context for up to `MAX_TURNS` turns.

This modular design separates concerns among transcription, language generation, text-to-speech, and state management, making the voice agent easy to understand and extend.
