JUSTI

Personal cooking assistant powered by local AI

Technical project documentation

FastAPI	Ollama · Mistral	XTTS v2	SQLite	Python 3

1. Project description

Justi is a conversational cooking assistant that runs entirely on-device, without relying on paid external APIs or sending user data to third parties. Users interact via text or voice and receive natural language responses generated by a local language model, accompanied by synthesized audio.

The project integrates three layers of artificial intelligence in a single application:

- Text generation with Ollama + Mistral (local LLM, 4.4 GB)
- Voice synthesis with XTTS v2 by Coqui (local Text-to-Speech)
- Speech recognition (Speech-to-Text) for microphone input
- Online recipe search via the Spoonacular API (with LLM fallback)
- Persistent per-user memory system stored in SQLite

Archivo wsl.localhost/

Hola, soy Justi

¿QUIERES SUBIR UNA RECETA O DOCUMENTO?

Subir

TU NOMBRE

joel

CHAT

dame una receta para el desayuno

Hola Joel, estoy encantado de poder ayudarte en la cocina! A continuación te presento una receta sencilla y no picante para tu desayuno: **Helado de Fruta**. ¡Es muy refrescante y te brindará energía para comenzar el día! Para prepararlo necesitarás los siguientes ingredientes: - Yogur sin grasa (1 taza - 240 ml) - Frutas frescas (tendrías que elegir las que prefieras, podrían ser fresas, peras, uvas, frambuesas, etc.) (2 tazas - 480 ml) - Miel o azúcar de caña al gusto (1 cucharadita - 5 ml) Pasa los ingredientes a través de un procesador o licuadora hasta obtener una mezcla cremosamente suave. A continuación, distribuiremos el helado en copas y lo pondremos en el congelador durante al menos 3 horas para que se solidifique. Listo! Una receta sencilla, fresca y muy nutritiva para tu desayuno. ¡Aprovecha y disfruta!

¿QUÉ NECESITAS?, ¿QUÉ INGREDIENTES TIENES?

Dame una receta para el desayuno con 3 huevos

Preguntar

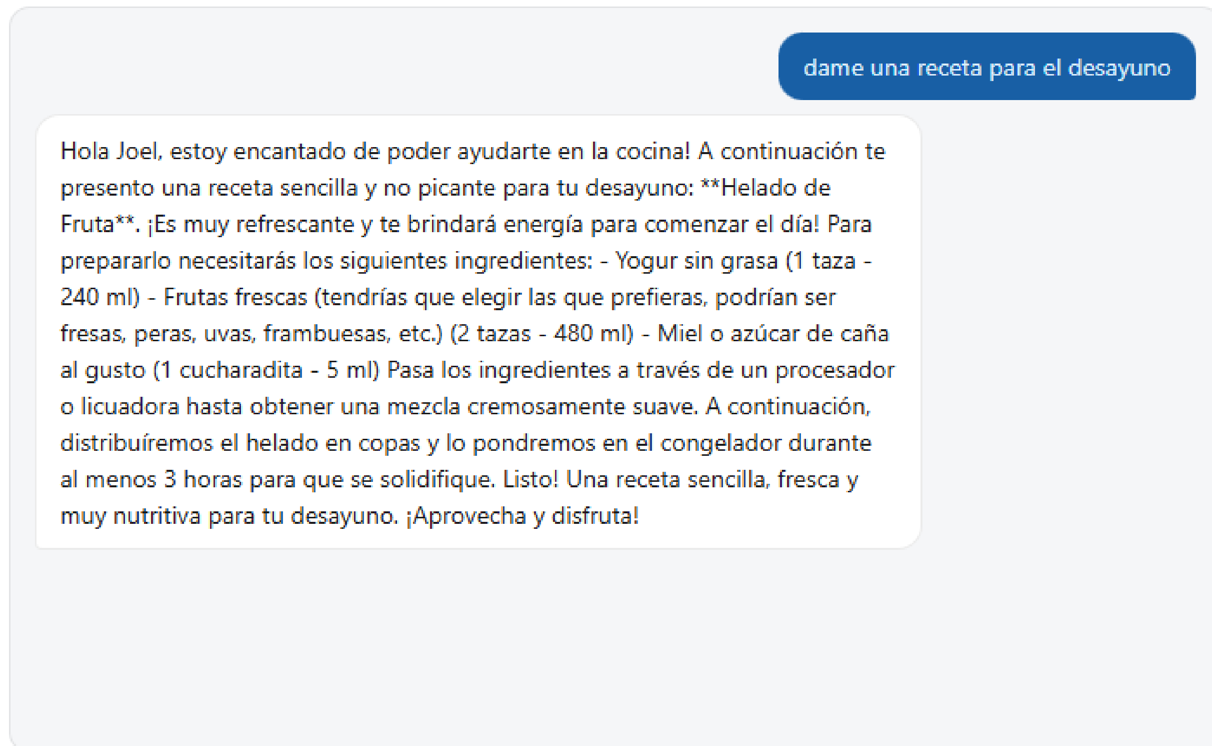
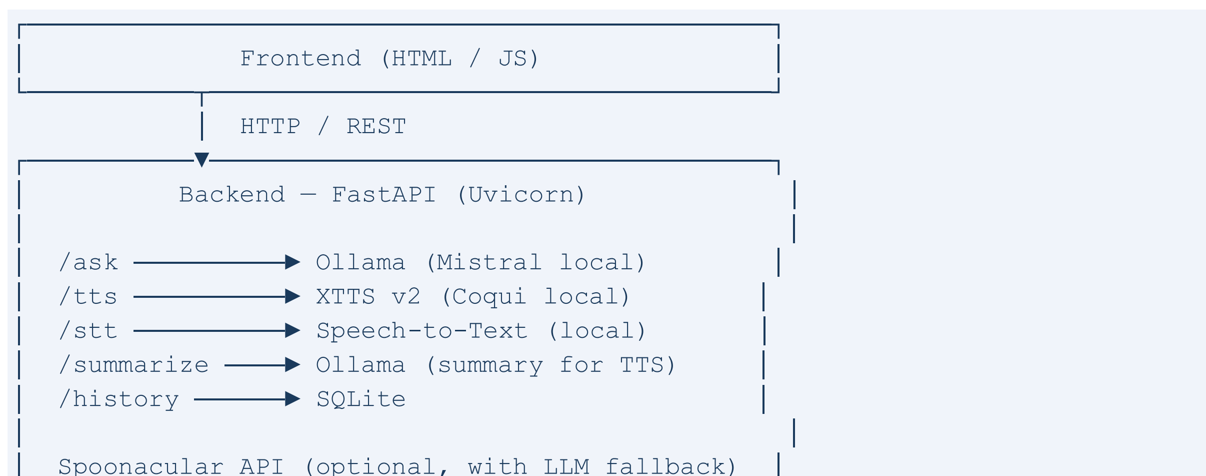


Figure 1 — Justi's main interface in the browser

---

2. General architecture

The application follows a single-layer client-server architecture. The frontend is a static HTML/JS page that communicates with the backend through fetch calls to the REST API.



```
│  SQLite DB: users, messages, memories, recipes  │
└─────────────────────────────────────────────────┘
```

All AI processing happens on the same machine. No cloud model calls are made during the normal conversation flow.

Figure 2 — Startup logs: Uvicorn running with all endpoints ready

---

3. API endpoints

The backend exposes 5 main endpoints. All of them accept and return JSON except /tts, which returns binary audio.

POST /ask

Main conversation endpoint. Implements a 5-step pipeline in priority order:

- Pending memory confirmation (responses 'yes' / 'no')
- Block if unconfirmed pending memory exists
- New memory extraction from the user's message
- Recipe search on Spoonacular (if the query requires it)
- Response with local LLM (Mistral via Ollama) with memory context

Method	**POST**
Route	`/ask`
Description	Generates a text response for the user's question.
Body / Input	`{ "user": "joel", "question": "give me a chicken recipe" }`

Response	`{ "answer": "...", "type": "message" | "confirm_request" | "recipe_list" }`

POST /tts

Receives text and returns a WAV audio file generated by XTTS v2 running locally.

Method	**POST**
Route	`/tts`
Description	Converts text to WAV audio using XTTS v2.
Body / Input	`{ "text": "Hi, here is your recipe..." }`
Response	`audio/wav (filename: justi.wav)`

```python
class TTSRequest(BaseModel):
    text: str

@app.post("/tts")
def tts_endpoint(req: TTSRequest):
    audio_bytes = text_to_speech_bytes(req.text)
    return Response(
        content=audio_bytes,
        media_type="audio/wav",
        headers={"Content-Disposition": "inline; filename=justi.wav"}
    )
```

Figure 3 — Implementation of the /tts endpoint

POST /stt

Receives an audio file uploaded by the user and returns its text transcription.

Method	**POST**
Route	`/stt`
Description	Converts user audio to text (Speech-to-Text).
Body / Input	`multipart/form-data → field: audio (UploadFile)`
Response	`{ "text": "give me a breakfast recipe" }`

```
@app.post("/stt")
async def stt_endpoint(audio: UploadFile = File(...)):
    audio_bytes = await audio.read()
    text = speech_to_text(audio_bytes)
    return {"text": text}
```

Figure 4 — Implementation of the /stt endpoint

POST /summarize-for-voice

Generates a 2-sentence short and natural summary to be read by TTS. Used internally before calling /tts when the LLM response is long.

Method	**POST**
Route	`/summarize-for-voice`
Description	Summarizes text into 2 sentences for voice synthesis.
Body / Input	`{ "text": "long LLM response..." }`
Response	`{ "summary": "Short 2-sentence summary." }`

GET /history/{user}

Method	**GET**
Route	`/history/{user}`
Description	Returns the user's message history in chronological order.
Body / Input	`— (no body)`
Response	`[ { "role": "user"|"assistant", "content": "...", "created_at": "..." } ]`

---

4. Language model — Ollama + Mistral

The LLM runs entirely locally using Ollama as the runtime. The chosen model is Mistral 7B in its quantized version (4.4 GB), which offers a good balance between response quality and RAM/VRAM consumption.

```
(venvReal) joel@joel:          $ ollama list
NAME              ID              SIZE      MODIFIED
mistral:latest    6577803aa9a0    4.4 GB    3 months ago
```

Figure 5 — Mistral model installed locally with Ollama

On every call to /ask, a dynamic system prompt is built that includes the user's name, dietary preferences, allergies, and diet — all retrieved from the memories database:

```
    memories = load_memories(user_id)
    memory_text = ""
    nombre = None

    for mem in memories:
        memory_text += f"- {mem['key']}: {mem['value']}\n"
        if mem["key"] == "nombre":
            nombre = mem["value"]

    system_prompt = f"""
u nombre es Justi, un asistente de cocina personal.

nformación conocida del usuario:
memory_text}

eglas:
 Usa esta información SIEMPRE que sea relevante.
 Si conoces el nombre del usuario, úsalo.
 No inventes datos.
 No repitas la memoria explícitamente.

stilo:
 Cercano, amable y claro.
 Explica recetas paso a paso.
""

    response = ollama.chat(
        model="mistral",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": req.question}
        ]
    )
```

Figure 6 — Dynamic system prompt with user memory and ollama.chat() integration

---

5. Memory system

Justi detects personal information in the user's messages using regular expressions and stores it in SQLite. Before saving any data, the system asks for explicit confirmation from the user (reply 'yes' or 'no').

Key	Description
name	User's name (unique, overwritten on update)
diet	Dietary regime: vegetarian, vegan, etc. (unique)

likes	Foods the user likes
dislikes	Foods the user dislikes
allergy	Food allergies or intolerances
favorite_food	Favorite food as stated by the user

The confirmation flow ensures that Justi does not save data due to misinterpretation. The 'name' and 'diet' keys are unique: if a new value is detected, the previous one is deleted before inserting.

```python
def extract_memories(text: str):
    memories = []
    t = text.lower().strip()

    basura = ["hola", "gracias", "ok", "jaja", "jeje"]
    if t in basura:
        return memories

    patterns = [
        (r"mi nombre es\s+([a-záéíóúñ ]+)", "nombre"),
        (r"me llamo\s+([a-záéíóúñ ]+)", "nombre"),

        (r"no me gusta[n]?\s+(.*)", "gusto_no"),
        (r"me gusta[n]?\s+(.*)", "gusto"),

        (r"soy\s+(vegetariano|vegano)", "dieta"),
        (r"soy alergico a\s+(.*)", "alergia"),

        (r"mi comida favorita es\s+(.*)", "comida_favorita"),
    ]

    for pattern, key in patterns:
        match = re.search(pattern, t)
        if match:
            memories.append((key, match.group(1).strip()))

    return memories
```

Figure 7 — extract_memories() function: preference detection with regex

6. Database — SQLite

Persistence is managed with SQLite through a custom module (db/database.py). No ORM is used — queries are plain SQL with Python's built-in sqlite3 module.

Table	Contents
users	id, name — one record per username

messages	id, user_id, role, content, created_at — full history
memories	id, user_id, key, value — confirmed user memory
pending_memories	id, user_id, key, value — memory awaiting confirmation
recipes	id, title, ingredients, instructions, source — saved recipes

7. Spoonacular integration

When the user's message contains keywords such as 'recipe', 'cook', or 'prepare', the system first attempts to fetch real recipes from the Spoonacular API before falling back to the LLM. The API key is loaded from a .env file using python-dotenv.

If Spoonacular returns no results or raises a SpoonacularError, the system automatically falls back to the LLM without the user noticing. Found recipes are saved to the recipes SQLite table for future queries.

Parameter	Source
query	Original user question
diet	User memory — 'diet' key
intolerances	User memory — 'allergy' key
API Key	Environment variable SPOONACULAR_API_KEY (.env)

8. Technologies and dependencies

Technology	Version / Usage
Python 3	Main backend language
FastAPI	Web framework for the REST API
Uvicorn	ASGI server with hot-reload for development
Ollama	Runtime for running Mistral locally
Mistral 7B	Language model (4.4 GB, quantized)
XTTS v2 (Coqui)	Local Text-to-Speech voice synthesis

SQLite + sqlite3	Local database, no server required
python-dotenv	Environment variable management (.env)
Spoonacular API	Online recipe search (optional)
HTML / CSS / JS	User interface — static client

---

9. Project highlights

- 100% local and private: Ollama, Mistral, and XTTS v2 run on the user's machine. No conversation data leaves the device.
- Multimodal AI pipeline: text, voice input (STT), and voice output (TTS) in a single application.
- Memory system with explicit confirmation: the user controls what information Justi stores.
- Intelligent fallback: if Spoonacular fails, the LLM responds without interrupting the conversation.
- REST API documented with FastAPI / Swagger UI accessible at /docs.
- No cloud model dependencies: the project works offline once the models are downloaded.