생성형 AI 기반 RAG 개발자 과정

챗봇 진화 — 대화 / 세션 / 영속화

Session 5 / 33 — Day 1

Flask UI · history · 토큰한도 · SQLite · session · 요약 · SSE

박수현 · (사)한국정보공학기술사회 · 2026

LLM 이론 · LangChain · ChromaDB CRUD · Agent — 4일 핸즈온

이번 시간을 마치면

핵심 (이 세션의 뼈대)

- ✅ **history 누적** 으로 stateless API 위에 대화 상태를 만든다 — *멀티턴의 전부*
- ✅ **Temperature** 로 출력의 정확함 ↔ 창의성을 조절한다
- ✅ **토큰 한도** 가 history 의 자연 상한 → **이력 관리**(window·요약)가 필요

확장 (서비스화 — 심화/옵션)

- □ **SQLite·세션** 으로 영속화·사용자별 격리
- □ **요약 압축** 으로 긴 대화의 비용·정확도를 잡는다
- □ **SSE 스트리밍** 으로 응답 체감 시간을 줄인다

실습

- ✅ `1.openai/2~5.chatbot*/` 6개 파일 — 단계별 실행
- ✅ `1.openai/6.twobots/` — 두 봇이 자동 대화
- ✅ `1.openai/8.streaming/` — SSE 스트리밍 챗봇

> 📂 베이스 소스: 모두 Flask 기반 단일 파일. 한 단계씩 진화하는 구조라 매 단계의 **diff 만 보면 됨**.

7단계로 진화시키는 챗봇

단계	베이스 코드	핵심 진화
1	`2.chatbot_ui/`	Flask + 정적 HTML + REST 호출 한 번
2	`3.chatbot2_history/app3_history.py`	`conversation_history` 리스트 누적
3	`3.chatbot2_history/app4_historylimit.py`	마지막 N개만 유지 (token window)
4	`4.chatbot3_historysqlite/app4_historysqlite3.py`	SQLite 로 영속화
5	`5.chatbot4_session/app5_session.py`	`session_id` 도입
6	`5.chatbot4_session/app7_session_summary.py`	요약으로 압축
7	`8.streaming/1.sse_stream.py`	SSE 청크 전송

> 💡 **1~3단계**(history·window·temperature)가 이 세션의 핵심 — 멀티턴 챗봇의 본질. **4~7단계**(SQLite·세션·요약·SSE)는 서비스화로 가는 **확장**이니, 시간이 빠듯하면 핵심을 확실히 잡고 나머지는 코드 위치만 짚고 넘어가도 됨.

PART A

PART A
1·2단계 — UI + history

> stateless 위에 대화를 얹는 가장 짧은 코드 — 약 15분

`2.chatbot_ui/app2_openailib.py` — 한 호출 챗봇

```
from flask import Flask, request, jsonify, send_from_directory
from openai import OpenAI
from dotenv import load_dotenv
import os

load_dotenv('../.env')
app = Flask(__name__, static_folder='public')
client = OpenAI(api_key=os.getenv('OPENAI_API_KEY'))

@app.route('/api/chat', methods=['POST'])
def chat():
    user_input = request.json.get('userInput', '')
    r = client.chat.completions.create(
        model='gpt-4o-mini',
        messages=[
            {'role': 'system', 'content': 'You are a helpful assistant.'},
            {'role': 'user',   'content': user_input},
        ],
    )
```

한계 — 매 요청이 독립

- "대한민국 수도는?" → "서울" → "그 도시 인구는?" 물으면 **"그 도시"가 어디인지 모름**
- Chat Completion API 는 **stateless** — 직전 대화가 다음 요청에 함께 전달되지 않음
- 모델 결함이 아니라 **호출 방식의 특성** — 맥락은 클라이언트(우리 코드)가 매번 함께 보내야 유지됨

`3.chatbot2_history/app3_history.py` — 메모리에 history

```
# 모듈 전역에 history 리스트 — 모든 사용자가 공유 (단점!)
conversation_history = []

@app.route('/api/chat', methods=['POST'])
def chat():
    user_input = request.json.get('userInput', '')

    # ① 사용자 메시지를 history 에 누적
    conversation_history.append({'role': 'user', 'content': user_input})

    # ② history 전체를 API 로 전송 (stateless 위의 대화 시뮬)
    r = client.chat.completions.create(
        model='gpt-4o-mini',
```

> 💡 **멀티턴의 전부는 이 3단계** — ① user 추가 ② 전체 messages 전송 ③ assistant 응답을 다시 추가. 모델의 지난 답변을 `assistant` 역할로 되넣어야 자기 답을 근거로 맥락을 이어감.

새로 생긴 문제 3가지

1. **모든 사용자가 같은 history 공유** — 다른 사람이 친 메시지가 보임
2. 서버 재시작하면 history 사라짐
3. 대화가 길어질수록 매 호출 토큰 폭증 → 비용 ↑

자연 상한 — context window

모델	컨텍스트 한도	의미
gpt-4o-mini	**128K** 토큰	약 한국어 책 한 권
gpt-4o	**128K**	동일
gpt-4.1	**1M** 토큰	책 7~10권
claude-haiku-4-5	**200K**	책 1.5권
claude-opus-4-7	**1M**	책 7~10권

충돌 시 에러

```
openai.BadRequestError: This model's maximum context length is 128000 tokens.
However, your messages resulted in 132847 tokens.
```

대응 3가지

1. **Window** — 최근 N개만 유지 (단순, 정보 손실 가능)
2. **Summary** — 오래된 대화를 요약으로 압축 (정보 보존, 약간의 비용)
3. **RAG** — history 를 벡터DB 에 넣고 관련 부분만 검색 (Day 2 주제)

`app4_historylimit.py` — 마지막 N개만

```
WINDOW_SIZE = 10   # 최근 10턴만 유지 (system 메시지는 별도)

@app.route('/api/chat', methods=['POST'])
def chat():
    user_input = request.json.get('userInput', '')
    conversation_history.append({'role': 'user', 'content': user_input})

    # system 메시지 + 최근 WINDOW_SIZE 만 API 로 전송
    system_msg = [{'role': 'system', 'content': '한국어 전문가.'}]
    recent = conversation_history[-WINDOW_SIZE:]
    messages = system_msg + recent

    r = client.chat.completions.create(model='gpt-4o-mini', messages=messages)
    reply = r.choices[0].message.content
```

윈도우 크기 선택

- **5턴** — 짧은 작업 (Q&A, 검색)
- **10턴** — 일반 챗봇
- **20턴+** — 멀티턴 추론 (코딩·기획)

> ⚠️ **system 메시지** 는 항상 윈도우 밖에서 별도 보존. 잘라먹으면 페르소나가 망가짐.

같은 질문, 다른 성격의 답

파라미터 한눈에

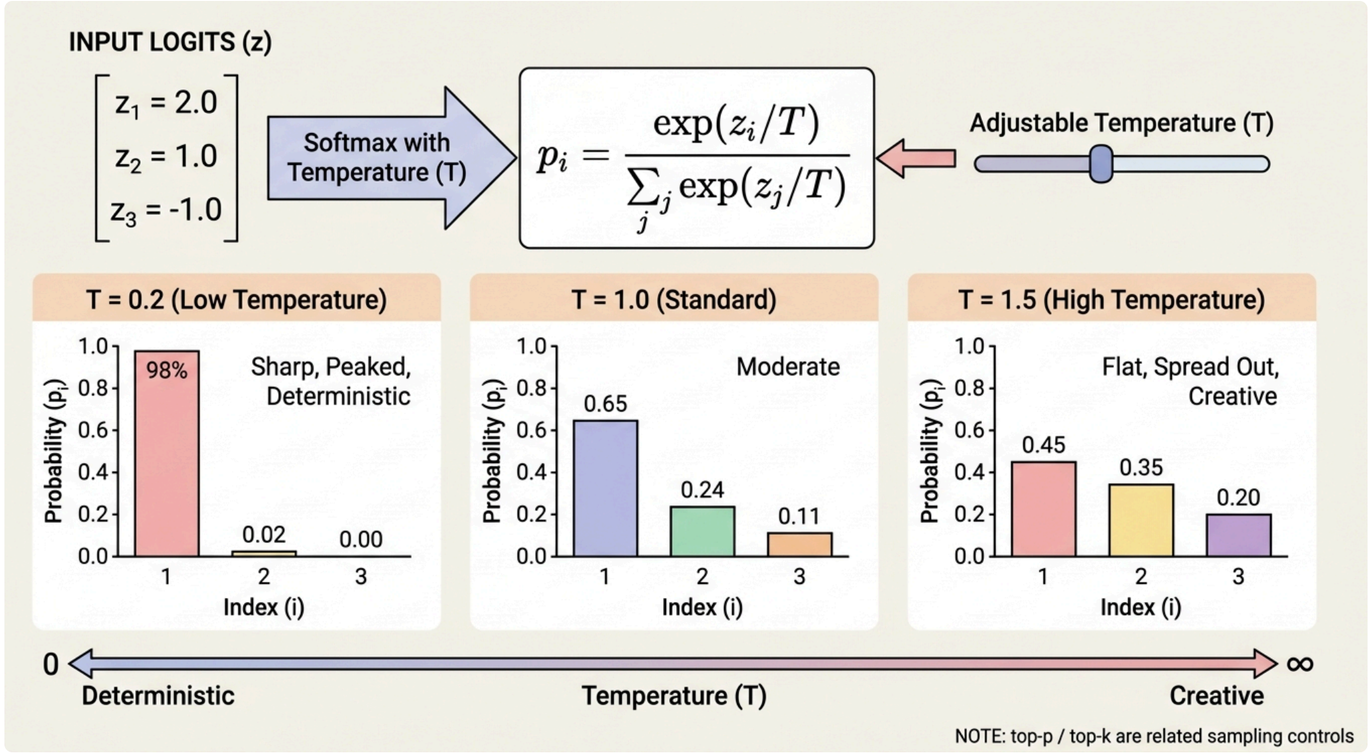
파라미터	역할	범위
temperature	무작위성·창의성	0.0 ~ 2.0
max_tokens	응답 최대 토큰	모델 한계 내
top_p	상위 확률 토큰만 후보	0.0 ~ 1.0
frequency_penalty	같은 단어 반복 억제	-2.0 ~ 2.0
presence_penalty	새 주제 도입 장려	-2.0 ~ 2.0

```
# 1.openai/1.intro/12.sdk_params.py
response = client.chat.completions.create(
    model='gpt-4o-mini',
    messages=[
        {'role':'system','content':'친절한 AI 도우미.'},
        {'role':'user','content':'AI를 한 문장으로 설명해줘.'},
    ],
    temperature=0.7,   # 0.0 정확·일관 ~ 2.0 창의·다양
    top_p=0.9,
)
```

- **낮게(≈0)** — RAG·사실 응답: 일관·재현성 ↑
- **높게** — 브레인스토밍·창작: 다양성 ↑

💡 일부 추론 특화 모델은 temperature 를 지원하지 않음 — 모델별 지원 파라미터 확인 후 적용. **Day 2 RAG 에서는 낮은 temperature 가 기본.**

Temperature — T 가 확률 분포의 뾰족함을 바꾼다



Temperature 샘플링 softmax(z_i/T) · T 낮으면 뾰족(결정적), 높으면 평평(창의·다양) · top-p/top-k와 함께 제어

PART B

PART B
4단계 — SQLite 영속화

> 서버 재시작에도 살아남는 대화 — 약 10분

`app4_historysqlite3.py` — DB 가 history

스키마 + insert/select 함수

```
conn = sqlite3.connect('history.db', check_same_thread=False)
conn.row_factory = sqlite3.Row
cursor = conn.cursor()

def init_db():
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS conversation (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            role TEXT NOT NULL,
            content TEXT NOT NULL,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
    """)
    conn.commit()

def insert(role: str, content: str):
    cursor.execute("INSERT INTO conversation (role, content) VALUES (?, ?)",
                   (role, content))
    conn.commit()

def get_recent(n: int = 20) -> list[dict]:
    cursor.execute(
        "SELECT role, content FROM conversation ORDER BY id DESC LIMIT ?", (n,))
    rows = cursor.fetchall()
    return [{'role': r['role'], 'content': r['content']} for r in reversed(rows)]
```

`chat()` 가 단순해진다

```
user_input = request.json.get('userInput', '')
insert('user', user_input)
messages = get_recent(20)
r = client.chat.completions.create(model='gpt-4o-mini', messages=messages)
reply = r.choices[0].message.content
insert('assistant', reply)
```

정리 — 4단계 효과

- 서버 재시작 후에도 **대화 영구 보존**
- DB 쿼리로 통계·검색 가능
- 다음 단계(세션 분리)의 기반
- `WHERE session_id = ?` 한 줄로 격리 가능해짐

Flask + SQLite 함정 3가지

함정	증상	해결
same_thread 검사	`ProgrammingError: SQLite objects created in a thread can only be used in that same thread`	`sqlite3.connect(..., check_same_thread=False)`
요청마다 새 connection	성능 저하 + 잠금 충돌	Flask `g` 객체에 connection 저장, `teardown_appcontext` 로 정리
트랜잭션 누락	데이터 사라짐	INSERT 후 `conn.commit()` 잊지 말 것

권장 패턴 — Flask `g` + teardown

```
from flask import g

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect('history.db')
    return db

@app.teardown_appcontext
def close_db(exc):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()
```

> 💡 **운영 환경**: SQLite 는 단일 프로세스용. 동시 접속이 많으면 **PostgreSQL/MySQL** 또는 **Redis** 로.

PART C

PART C
5단계 — 세션 분리

> 누구의 대화인가 — 약 10분

4단계의 함정 — 모든 사용자가 같은 history

시나리오

1. A 가 "내 비번은 abc123" 입력 (실제 사고 사례!)
2. B 가 같은 챗봇 접속
3. B 가 "비번 알려줘" → 봇이 abc123 흘림 😱

해결 — `session_id` 라는 한 컬럼

```
CREATE TABLE conversation (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    session_id TEXT NOT NULL,        -- ← 추가
    role TEXT NOT NULL,
    content TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

세션 식별 방법 3가지

1. **쿠키** (Flask `session`) — 브라우저 기준
2. **JWT** — 토큰 기반 (모바일/SPA)
3. **URL 파라미터** — 데모용 (보안 약함)

`app5_session.py` — session_id 로 격리

세션 발급 + DB 함수 (sid 추가)

```
from flask import session
import uuid

app.secret_key = 'change-me-in-prod'

@app.before_request
def ensure_session_id():
    if 'sid' not in session:
        session['sid'] = str(uuid.uuid4())

def insert(sid: str, role: str, content: str):
    cursor.execute(
        "INSERT INTO conversation "
        "(session_id, role, content) VALUES (?, ?, ?)",
        (sid, role, content))
    conn.commit()

def get_recent(sid: str, n: int = 20) → list[dict]:
    cursor.execute(
        "SELECT role, content FROM conversation "
        "WHERE session_id = ? ORDER BY id DESC LIMIT ?",
        (sid, n))
    return [{'role':r['role'],'content':r['content']}
            for r in reversed(cursor.fetchall())]
```

`chat()` — sid 로 격리

```
@app.route('/api/chat', methods=['POST'])
def chat():
    sid = session['sid']
    user_input = request.json.get('userInput', '')
    insert(sid, 'user', user_input)
    messages = get_recent(sid, 20)
    r = client.chat.completions.create(
        model='gpt-4o-mini', messages=messages)
    reply = r.choices[0].message.content
    insert(sid, 'assistant', reply)
    return jsonify({'reply': reply})
```

> 💡 **세션 삭제 엔드포인트** (`app6_session_delete.py`) 도 추가 — 사용자가 "내 대화 지우기" 기능을 가질 수 있게.

PART D

PART D
6단계 — 요약 압축

> 긴 대화를 압축해 토큰을 줄이기 — 약 10분

window 만으로 부족한 이유

사례 — 30턴 대화 (코딩 페어 작업)

- 1~10턴: 요구사항 정의
- 11~20턴: 아키텍처 토론
- 21~30턴: 구체 구현

Window=10 으로 잘라먹으면

- 21~30턴만 남고 **1~20턴의 요구·아키텍처가 사라짐**
- 봇이 갑자기 다른 답 → 사용자 혼란

Summary 패턴

1. N턴 (예: 20) 이상 쌓이면
2. **오래된 절반을 요약**해 1개의 system 메시지로 압축
3. 최근 N/2 턴은 그대로 유지
4. 결과 — 토큰 절약 + 정보 보존

```
[system] 한국어 전문가
```

app7_session_summary.py 핵심

요약·압축 함수

```
SUMMARY_TRIGGER = 20   # 20턴 이상이면 요약
KEEP_RECENT = 10       # 최근 10턴은 유지

def summarize_old(messages: list[dict]) → str:
    """오래된 메시지들을 한 문단으로 압축."""
    old_text = "\n".join(
        f"{m['role']}: {m['content']}" for m in messages)
    r = client.chat.completions.create(
        model='gpt-4o-mini',
        messages=[
            {'role':'system','content':
             '다음 대화를 한 문단으로 핵심만 요약하세요.'},
            {'role':'user',  'content': old_text},
        ],
        max_tokens=300,
        temperature=0.0,
    )
    return r.choices[0].message.content
```

압축 트리거 + 적용

```
def compress_history(sid: str):
    rows = get_all_messages(sid)
    if len(rows) ⩽ SUMMARY_TRIGGER:
        return   # 아직 압축 불필요
    old = rows[:-KEEP_RECENT]   # 오래된 부분
    summary = summarize_old(old)
    delete_messages_before(
        sid, rows[-KEEP_RECENT]['id'])
    insert(sid, 'system',
           f'(이전 대화 요약) {summary}')
```

트리거 시점

- N턴 도달 시 (예: 매 20턴)
- 토큰 추정치가 한도의 80% 도달 시
- 사용자가 명시적으로 "정리해줘" 요청 시

PART E

PART E
보너스 — 두 봇 대화 (twobots)

> 자동 토론·면접·페어 코딩 — 약 8분

6.twobots/1.debate — 두 페르소나 토론

두 페르소나 + speak 함수

```
client = OpenAI()

PERSONA_A = "당신은 강력한 AI 규제를 주장하는 윤리학자."
PERSONA_B = "당신은 AI 혁신 자유를 옹호하는 기술 기업가."

def speak(persona: str, history: list[dict]) → str:
    r = client.chat.completions.create(
        model='gpt-4o-mini',
        messages=[{'role':'system','content':persona}]
                 + history,
        max_tokens=200,
        temperature=0.7,
    )
    return r.choices[0].message.content
```

10턴 자동 토론 루프

```
# A 의 첫 발언
topic = "AI 모델 학습에 저작권 콘텐츠 사용을 금지해야 하는가?"
history = [{'role':'user','content':f'주제: {topic}'}]
say_a = speak(PERSONA_A, history)
history.append({'role':'assistant','content':say_a})

for turn in range(10):
    # B 가 응답 (B 시점에서 A 발언이 user)
    history_b = swap_roles(history)
    say_b = speak(PERSONA_B, history_b)
    history.append({'role':'user','content':say_b})

    # A 가 응답
    say_a = speak(PERSONA_A, history)
    history.append({'role':'assistant','content':say_a})

    print(f"[Turn {turn}] A: {say_a[:80]}…")
    print(f"[Turn {turn}] B: {say_b[:80]}…")
```

응용

- 면접 봇 · 페어 코딩 · 평가자
- Day 4 **Evaluator-Optimizer** 의 기초

PART F

PART F
7단계 — SSE 스트리밍

> 응답 체감 시간을 줄이는 기법 — 약 12분

사용자 체감 = "첫 글자가 보이는 시간"

❌ 일반 응답 (non-streaming)

```
사용자 입력
  ↓ 0초
[기다림 ......]
  ↓ 5초 (생성 완료)
응답 전체가 한 번에 등장
```

- 5초 동안 화면 정지
- 사용자: "안 되는 건가?"

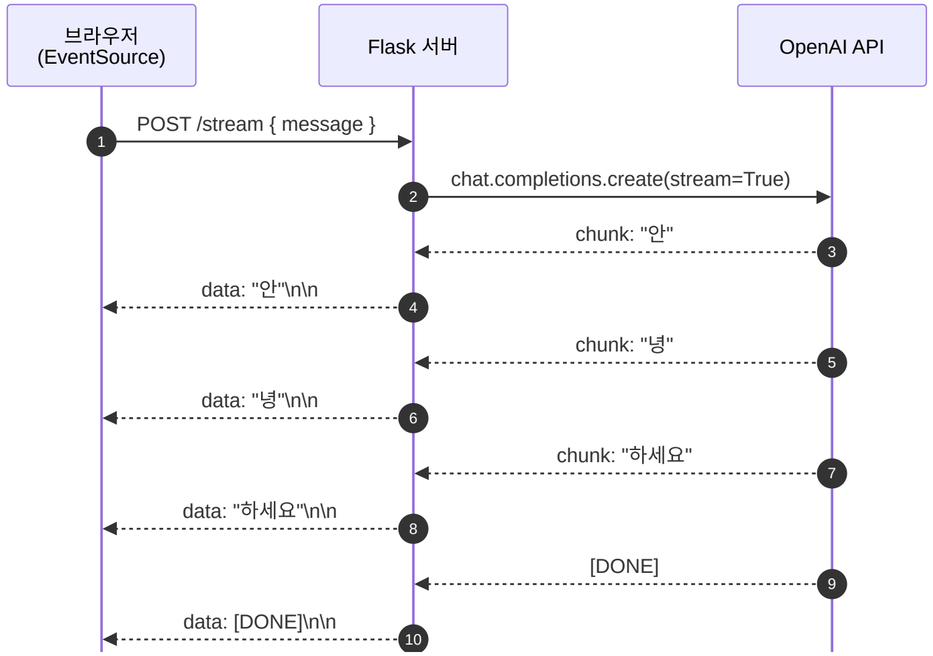
✅ 스트리밍 (streaming)

```
사용자 입력
  ↓ 0초
[기다림 .....]
  ↓ 0.5초 → "안" 등장
  ↓ 0.6초 → "안녕"
  ↓ 0.8초 → "안녕하세요"
  ↓ ...
  ↓ 5초    → 전체 완료
```

- 첫 글자 **0.5초**
- 사용자: "오 응답 중이구나"

> 🎯 같은 5초 응답이라도 **체감 시간은 1/10**. ChatGPT 가 한 글자씩 타이핑되는 그 효과.

SSE — 서버가 클라이언트에 청크 푸시



SSE vs WebSocket — 챗봇에는 SSE

항목	SSE	WebSocket
방향	서버 → 클라이언트 (단방향)	양방향
프로토콜	HTTP/1.1	upgrade 후 별도
재연결	브라우저 자동	직접 구현
챗봇 적합도	⭐⭐⭐⭐⭐	⭐⭐⭐

챗봇에는 SSE 가 일반적

- 사용자 입력은 보통 단일 POST 로 충분
- 응답만 청크 단위 push → SSE 단방향 모델이 정확히 fit
- nginx/CDN 호환성도 HTTP/1.1 라 무난

`8.streaming/1.sse_stream.py` 핵심

```
from flask import Flask, Response, request

@app.route('/stream', methods=['POST'])
def stream():
    user_message = request.json.get('message', '')

    def generate():
        response = client.chat.completions.create(
            model='gpt-4o-mini',
            messages=[
                {'role':'system','content':'한국어 친절 도우미.'},
                {'role':'user',  'content': user_message},
            ],
            stream=True,                          # ← 핵심!
            temperature=0.7,
            max_tokens=1000,
        )
        for chunk in response:                    # 청크 단위 iteration
            delta = chunk.choices[0].delta.content
            if delta:
                # SSE 포맷: "data: <payload>\n\n"
```

Flask 응답 헤더 자동 설정

- `mimetype='text/event-stream'` → SSE 인식
- 운영 환경에서는 `X-Accel-Buffering: no` 헤더 권장 (nginx 버퍼링 방지)

브라우저 측 — vanilla JS

HTML + send() 진입점

```
<input id="msg" />
<button onclick="send()">Send</button>
<div id="out"></div>

<script>
async function send() {
  const msg = document.getElementById('msg').value;
  const out = document.getElementById('out');
  out.textContent = '';

  // Fetch 로 POST + stream 응답 읽기
  const resp = await fetch('/stream', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify({message: msg}),
  });
```

스트림 청크 읽기·파싱

```
  const reader = resp.body.getReader();
  const decoder = new TextDecoder();
  while (true) {
    const {done, value} = await reader.read();
    if (done) break;
    const text = decoder.decode(value);
    // "data: {...}\n\n" 형식 파싱
    text.split('\n\n').forEach(line => {
      if (!line.startsWith('data: ')) return;
      const payload = line.slice(6);
      if (payload === '[DONE]') return;
      const {token} = JSON.parse(payload);
      out.textContent += token;
    });
  }
}
</script>
```

> 💡 React/Vue 사용 시 동일 패턴 — `ReadableStream` reader 로 청크 읽기. 라이브러리 따로 필요 없음.

PART G

PART G
미니 실습

대상 도메인 챗봇 — 약 25분

실습 과제 (25분)

1단계 — 대상 도메인 system 메시지 작성 (5분)

- 예: "부동산 법률 전문가 / 사내 코딩 컨벤션 봇 / 학과 과제 도우미"

2단계 — `5.chatbot4_session/app7_session_summary.py` 를 대상 도메인에 맞게 수정 (15분)

- system 메시지 변경
- 토픽 일관성 확인 — 10턴 이상 대화하며 요약 트리거 발동되는지

3단계 — SSE 로 업그레이드 (5분, 옵션)

- `8.streaming/` 의 SSE 패턴 가져와 위 챗봇에 스트리밍 적용

결과 보고 — 한 줄 요약

- 만든 챗봇에서 가장 잘 답한 질문 / 가장 빗나간 질문
- Day2 (Session 9) 에서 LangChain 으로 다시 만들 때 비교 자료로 쓸 것

> 📌 답 안 나오면 `system` 메시지를 더 구체적으로 (역할 + 톤 + 금지 사항).

핵심 8가지

1. ✅ **stateless API** 위에 챗봇을 만든다 = **messages 배열을 클라이언트가 관리**
2. ✅ 메모리 → SQLite → session_id → 요약 — **7단계 진화** 패턴
3. ✅ Context window 한도가 history 의 자연 상한 (gpt-4o-mini: 128K)
4. ✅ **Temperature** = 정확함 ↔ 창의성 다이얼 (RAG 는 낮게, 창작은 높게)
5. ✅ Window 만으론 정보 손실 → **요약 압축** 으로 정보 보존 + 토큰 절감
6. ✅ Flask + SQLite 함정 3가지 (thread, connection, commit)
7. ✅ 두 봇 대화 패턴 — Evaluator-Optimizer 류 멀티에이전트의 기초
8. ✅ **SSE** = 같은 응답을 10배 빠르게 느끼게 하는 기법

실습 과제 — 직접 해보기

필수

- [ ] `1.openai/3~5.chatbot*/` 6개 파일 모두 실행해 챗봇 띄우기
- [ ] 미니 실습 — 대상 도메인 챗봇 + system 메시지 작성

옵션

- [ ] `1.openai/8.streaming/` 의 SSE 챗봇 띄우고 응답 체감 비교
- [ ] **LangChain 패키지** 설치 확인: `bash python -c "import langchain, langchain_openai; print('ok')"`

📤 정리

- 만든 챗봇 스크린샷 1장 + system 메시지 한 단락 — Slack/메일