목표: RAG 사용하는 챗봇 만들기

v0.1 기초공사

단계	작업 내용	도구/기술	설명
1	`docker-compose`로 vLLM 실행	**vLLM, gemma**	LLM 서버(gemma)를 vLLM으로 배포하고 inference API로 노출
2	문서 → embedding → DB 저장 기능 구현	**BGE-m3-ko, ChromaDB**	문서를 임베딩 후 벡터 DB(ChromaDB)에 저장하여 검색 가능하게 구성
3	질문 → 검색 → 답변 생성	**RAG 파이프라인**	사용자 질문에 대해 벡터 검색 후 context를 기반으로 LLM 응답 생성
4	FastAPI 서버 통합	**FastAPI, API 라우터 구성**	문서 업로드, 질문 처리 등 API 라우팅을 FastAPI 기반으로 구현
5	세션 관리 기능 개선	**Redis**	사용자별 세션 이력 관리 기능을 Redis 기반으로 리팩터링

1. vLLM docker-compose 로 구성하기

1) docker-compose 생성 및 실행

docker-compose.yml 생성

`/src/images/vllm/docker-compose.yml`

실행 명령

```
cd src/images/vllm
docker compose up -d
```

테스트 (모델 로딩 확인)

```
curl http://localhost:48000/v1/models
```

```
# 로그 확인 명령어
docker logs vllm-server-sjchoi --tail 100
```

모델 로딩 완료 확인을 위한 로그 모니터링

```
docker logs -f vllm-server-sjchoi
```

→ 로딩이 끝나면 아래와 비슷한 로그가 출력됩니다:

```
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

---

2. 2단계: fastAPI

2_1. docker-compose 실행 및 문서 임베딩(임시) → 벡터 DB(ChromaDB) 저장 기본 코드 작성

1) 소스코드

항목

항목	구성
임베딩 모델	`BGE-m3-ko` (HuggingFace)
벡터 DB	`ChromaDB` (Python 기반 로컬 DB)
인터페이스	FastAPI (문서 업로드 + 임베딩 트리거)
Docker 경로	`/src/images/fastapi/docker-compose.yml`
소스코드	`/src/volumns/fastapi` 안에 구성

디렉토리 구성

```
/src/volumns/fastapi
├── app/
│   ├── main.py             ← FastAPI 진입점
│   ├── embed.py            ← 문서 분할 + 임베딩
│   ├── chroma_db.py        ← ChromaDB 연결
│   └── model_loader.py     ← BGE-m3-ko 로딩
└── requirements.txt        ← transformers + chromadb 등
```

```
# 파일/디렉토리 생성 명령어
mkdir -p volumns/fastapi/app

touch volumns/fastapi/app/main.py
touch volumns/fastapi/app/embed.py
touch volumns/fastapi/app/chroma_db.py
```

```
touch volumns/fastapi/app/model_loader.py
touch volumns/fastapi/requirements.txt
```

2) docker-compose 생성 및 실행

docker-compose.yml + Dockerfile 생성

`/src/images/fastapi/docker-compose.yml /src/images/fastapi/Dockerfile`

실행 명령

```
cd src/images/fastapi
docker compose up --build -d        # --build 는 한번만 실행
```

☑ 언제만 --build가 필요하냐면?

- 처음 이미지를 만들 때
- requirements.txt를 수정한 경우
- Dockerfile을 수정한 경우

테스트

1. Swagger UI 확인

```
curl http://localhost:48001/docs
```

2. 테스트 파일 생성 및 실행

```
echo "한국 수도는 서울입니다." > /labs/docker/images/chat-dev-
sjchoi/src/volumns/fastapi/test_doc.txt
```

```
curl -X POST -F "file=@/labs/docker/images/chat-dev-
sjchoi/src/volumns/fastapi/test_doc.txt" http://localhost:48001/upload-doc
```

2_2. 임베딩 모델 붙이기: BGE-m3-ko

임베딩 모델

`/labs/docker/volumes/ml-dev/share/model/BGE-m3-ko` 경로에 다운로드해둔 모델 사용

```
# docker-compose.yml 을 비롯한 소스코드 수정 후 테스트
cd /labs/docker/images/chat-dev-sjchoi/src/images/fastapi
docker compose down
docker compose up --build -d  # ← requirements.txt 수정 시 반드시 --build
```

2_3. 질문 기반 검색 API 만들기

새로운 API 엔드포인트 추가

- POST /search-doc
- 입력: JSON { "query": "질문 내용" }
- 출력: 관련 문서 리스트

1. `search_doc.py` 모듈 생성

```
touch volumns/fastapi/app/search_doc.py
```

2. `search_doc.py` 에 search_similar_docs 함수 생성
3. `main.py` 에 API 추가
4. 테스트

```
curl -X POST http://localhost:48001/search-doc \
     -H "Content-Type: application/json" \
     -d '{"query": "한국의 수도에 대해서 알아?"}'
```

☑ 만약 결과값이 없는 경우, 저장된 벡터값이 없어서일 수 있으므로 새로이 벡터값 저장해두기

```
curl -X POST -F "file=@/labs/docker/images/chat-dev-
sjchoi/src/volumns/fastapi/test_doc.txt" http://localhost:48001/upload-doc
```

2_4. ChromaDB 파이프라인에 파일명 기반 메타데이터 저장 추가

1. `main.py` 수정 (파일명 전달)
2. `embed.py` 수정 (metadata 저장)
3. 테스트

```
curl -X POST -F "file=@/labs/docker/images/chat-dev-
sjchoi/src/volumns/fastapi/test_doc.txt" http://localhost:48001/upload-doc
```

---

3. RAG 파이프라인

3_1. 세션 기반 질문-응답 API 구성 (LLM 호출 포함)

1. `app/services/chat_service.py` 추가

```
touch volumns/fastapi/app/services/chat_service.py
```

2. `main.py` 수정
3. `llm_servcie.py` 추가

```
touch volumns/fastapi/app/services/llm_servcie.py
```

4. 테스트

```
curl -X POST http://localhost:48001/chat \
  -H "Content-Type: application/json" \
  -d '{
        "session_id": "test-session-001",
        "query": "한국의 수도에 대해서 알아?",
        "top_k": 2
      }'
```

3_2. LLM 서버 연동 (vLLM)

1) LLM 서버 연동

1. 공통 네트워크 만들기 (한 번만 실행)

> 다른 docker-compose.yml을 통해 서버 관리하기 위해서 FastAPI와 vLLM 컨테이너를 동일한 Docker 네트워크에 수동으로 연결

```
docker network create rag-net
```

2. 각 docker-compose.yml에서 네트워크 명시

☑ FastAPI - /src/images/fastapi/docker-compose.yml

```
services:
  fastapi:
    ...
    networks:
      - rag-net

networks:
```

```
  rag-net:
    external: true
```

☑ vLLM - /src/images/vllm/docker-compose.yml

```
services:
  vllm:
    ...
    networks:
      - rag-net

networks:
  rag-net:
    external: true
```

3. vLLM 서버 실행 확인

```
curl http://localhost:48000/v1/models
# 정상일 경우 결과: {"data": [{"id": "gemma-3-12b-it", "object": "model", ...}]}
```

4. .env 파일에 vLLM 주소 설정

5. 도커 재실행

```
cd /labs/docker/images/chat-dev-sjchoi/src/images/fastapi
docker compose build
docker compose up -d
```

```
cd /labs/docker/images/chat-dev-sjchoi/src/images/vllm
docker compose down
docker compose up -d
```

6. vLLM 서버 접속 확인을 위한 "/llm-status" API 추가

- llm_service.py
- chat_service.py
- main.py

7. 테스트

```
curl http://localhost:48001/llm-status
# 성공: {"status":"ok","message":"LLM 서버 연결 성공"}
# 실패: {"detail":"LLM 서버에 연결할 수 없습니다."}
```

8. 실제 모델 정보 확인용 `/llm-status/detail` API 확장
    - `llm_service.py`
    - `chat_service.py`
    - `main.py`
9. 테스트

```
curl http://localhost:48001/llm-status/detail
```

2) LLM을 통해 결과받기

1. `requirements.txt` 수정 => requests 추가 변경 후 docker-compose build 필요

```
cd /labs/docker/images/chat-dev-sjchoi/src/images/fastapi
docker compose build
docker compose up -d
```

2. `llm_service.py` 의 call_llm() 메소드 수정
3. `.env` 파일 : DEFAULT_MODEL 등록
4. 테스트

```
curl -X POST http://localhost:48001/chat \
  -H "Content-Type: application/json" \
  -d '{"query": "한국의 수도에 대해서 알아?", "history": []}'
```

3_3. 대화 흐름(세션) 관리

> 2단계 키 (user_id + session_id) 기반 세션 관리 구조로 설계

☑ 세션 구조 설계 (유저별 세션)

```
# 세션 저장 구조
_session_history = {
    "user_id_1": {
        "session_1": [ {"role": "user", "content": "..."}, {"role": "assistant",
"content": "..."} ],
        "session_2": [...]
    },
    "user_id_2": {
        ...
```

```
    }
}
```

> session_store.py | 저장소 (데이터 저장/조회 책임) : 현재는 in-memory dict, 나중에 Redis 등으로 교체 가능 session_service.py | 비즈니스 로직 (저장소를 활용해 세션 흐름 제어)

1. 세션 저장소 구현 `stores/session_store.py` : 추후 Redis로 대체 가능하도록 구조화 - (dict[user_id][session_id] → list[message])

- get_history()
- add_message()

2. 세션 서비스 로직 `services/session_service.py` 추가

- get_history(user_id, session_id) → 기존 대화 불러오기
- append_history(user_id, session_id, role, content) → 메시지 추가
- chat_with_session(user_id, session_id, query) → ① 세션 히스토리 조회 ② 벡터 검색 문서 추가 ③ 프롬프트 구성 → LLM 호출 ④ 응답 저장 ⑤ 응답 반환

3. chat API 연결

- `main.py` - `/chat-session API` 생성

4. 테스트

```
curl -X POST http://localhost:48001/chat-session \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": "user123",
    "session_id": "sess001",
    "query": "내일 아침식사사 추천해줘"
  }'
curl -X POST http://localhost:48001/chat-session \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": "user123",
    "session_id": "sess001",
    "query": "가벼운거"
  }'
```

---

4. 소스코드 개선

4_1. logger 생성

1. 폴더 및 파일 생성

```
# src/volumns/fastapi/app 으로 이동

# 1. utils 디렉토리 생성
mkdir -p utils

# 2. 로거 등록
touch utils/logger.py
```

2. `main.py` 에 `logger` 광역 등록

3. 기존 print() 를 logger.info() 로 변경

4_2. router 분리

1. 폴더 구조 정리

```
/src/volumns/fastapi/app/
├── main.py                  ← FastAPI 앱 실행만 담당
├── routers/
│   ├── chat_router.py       ← /chat, /chat-session 등
│   ├── doc_router.py        ← /upload-doc
│   ├── search_router.py     ← /search-doc
│   └── llm_router.py        ← /llm-status, /llm-status/detail
├── services/
├── stores/
└── ...
```

2. 폴더 및 파일 생성

```
# src/volumns/fastapi/app 으로 이동

# 1. routers 디렉토리 생성
mkdir -p routers

# 2. 빈 __init__.py 추가 (Python 패키지로 인식되도록)
touch routers/__init__.py

# 3. 각 라우터 파일 생성
touch routers/chat_router.py
touch routers/doc_router.py
touch routers/search_router.py
touch routers/llm_router.py
```

3. 이동한 router 테스트

```
# FastAPI 서버 실행 확인
curl http://localhost:48001

# Swagger 문서
curl http://localhost:48001/docs

# /llm-status
curl http://localhost:48001/llm-status

# /llm-status/detail
curl http://localhost:48001/llm-status/detail

# /upload-doc
curl -X POST http://localhost:48001/upload-doc \
  -F "file=@/labs/docker/images/chat-dev-sjchoi/src/volumns/fastapi/test_doc.txt"

# /search-doc
curl -X POST http://localhost:48001/search-doc \
  -H "Content-Type: application/json" \
  -d '{"query": "세상에서 가장 큰 나라의 수도는?"}'

# /chat
curl -X POST http://localhost:48001/chat \
  -H "Content-Type: application/json" \
  -d '{"query": "한국의 수도는?", "history": []}'

# /chat-session
curl -X POST http://localhost:48001/chat-session \
  -H "Content-Type: application/json" \
  -d '{"user_id": "user123", "session_id": "session001", "query": "부산은 어디에 있어?"}'
```

4_3. Chroma DB 데이터 관리

1. `/src/images/fastapi/docker-compose.yml` 수정

- volumn 추가

```
volumes:
  - /labs/docker/volumes/chat-dev-sjchoi/src/volumns/chroma_db:/chroma_db # 추가
```

2. chroma 저장방식 변경

1. 만약 chroma.persist() 오류 발생 시,

`embed_service.py` 수정하여 chroma.persist() 추가해봄

```
chroma.persist()
```

AttributeError: 'Client' object has no attribute 'persist' 에러는 사용하고 계신 chromadb 라이브러리의 최신 버전에서 client.persist() 메서드가 더 이상 사용되지 않기 때문에 발생

2. 최신 방식인 PersistentClient 사용

- 최신 버전에서는 클라이언트를 생성할 때 persist_directory (또는 path)를 지정하면, 데이터 추가/삭제 시 자동으로 디스크에 변경 사항이 저장되도록 방식이 변경됨
- persist() 메서드 자체가 라이브러리에서 제거됨

- `chroma_db.py` 아래 내용으로 수정

```
# chroma_db.py
def get_chroma_client():
    # PersistentClient를 사용하여 클라이언트를 생성합니다.
    # 이 방식이 최신 버전에서 권장하는 방법입니다.
    return chromadb.PersistentClient(path="/chroma_db")
```

- `embed_service.py` 에 chroma.persist() 사용했다면 전부 삭제

3. docker compose 재실행

```
cd src/images/fastapi
docker-compose down
docker-compose up -d --build
```

3. 데이터 생성 테스트

```
# 1) 데이터 생성 /upload-doc
curl -X POST http://localhost:48001/upload-doc \
  -F "file=@/labs/docker/images/chat-dev-sjchoi/src/volumns/fastapi/test_doc.txt"

# 2) docker 다시 재시작
cd src/images/fastapi
docker-compose down
docker-compose up -d --build

# 3) /search-doc
curl -X POST http://localhost:48001/search-doc \
  -H "Content-Type: application/json" \
  -d '{"query": "세상에서 가장 큰 나라의 수도는?"}'

# 4) "documents":[[]] 내용이 있으면 데이터가 저장되는 것
```

---

5. 세션 관리 기능 개선: Redis 기반 저장 방식으로 치환

> session_store.py에서 기존에 사용하던 메모리 기반 세션 저장을 Redis 기반 저장 방식으로 치환

1. redis 정보 추가

1. `/src/images/fastapi/docker-compose.yml` 에 redis 추가

```
  redis:
    image: redis:7.2
    container_name: redis-server
    ports:
      - "48009:6379"  # 로컬에서 확인할 수 있게 노출
    volumes:
      - /labs/docker/images/chat-dev-sjchoi/src/volumns/redis:/data
    networks:
      - rag-net
```

2. requirement.txt 에 redis 추가

3. .env 에 Radis 설정 추가

```
# Redis 설정
REDIS_HOST=redis
REDIS_PORT=48009
REDIS_DB=0
REDIS_SESSION_TTL=3600
```

2. docker compose 재실행

```
cd src/images/fastapi
docker-compose down
docker-compose up -d --build
```

3. 테스트

```
curl -X POST http://localhost:48001/chat-session \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": "user123",
    "session_id": "sess001",
    "query": "내일 아침식사 추천해줘"
  }'
curl -X POST http://localhost:48001/chat-session \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": "user123",
    "session_id": "sess001",
```

```
    "query": "가벼운거"
  }'
```

- 문제 발생 및 해결

문제_1. 원격 Docker 데몬(ssh://redsoft@192.168.0.251)에 연결을 시도하다가 실패

```
# 에러 메세지
unable to get image 'fastapi-fastapi': error during connect: Get
"http://docker.example.com/v1.47/images/fastapi-fastapi/json": command [ssh -o
ConnectTimeout=30 -T -l redsoft -- 192.168.0.251 docker system dial-stdio] has
exited with exit status 255, make sure the URL is valid, and Docker 18.09 or later
is installed on the remote host: stderr=ssh: connect to host 192.168.0.251 port
22: Connection timed out

# 주요 에러 메세지
ssh: connect to host 192.168.0.251 port 22: Connection timed out
```

- 원인:
  - DOCKER_HOST=ssh://redsoft@192.168.0.251 환경 변수로 인해 Docker 명령이 모두 원격 서버로 전송되고 있음
  - 하지만 SSH 연결이 불가능하여 Docker 자체가 동작하지 않음
- 해결: 로컬 Docker로 전환 (일시적)
  - 일시적으로 DOCKER_HOST 해제
  - 이 방법은 현재 쉘 세션에만 적용

```
unset DOCKER_HOST
docker-compose down
docker-compose up -d --build
```

---

v0.2

단계	작업 내용	도구/기술	설명
A	예외 핸들러		예외 핸들러 별도 파일로 관리
6	세션 선택/삭제/초기화 기능	**FastAPI, Redis**	사용자 세션을 리스트업하거나 초기화/삭제할 수 있는 API 추가

단계	작업 내용	도구/기술	설명
7	업로드 문서 관리 기능	**FastAPI, ChromaDB**	업로드한 문서 조회 및 제거하는 기능
B	세션 및 챗 서비스 구조 리팩터링		세션
8	문서 업로드 & 임베딩 구조 정리	**FastAPI, ChromaDB, sentence-transformers, PyMuPDF, python-docx, bs4**	다양한 입력 소스(txt, pdf, docx, URL)를 수용할 수 있도록 파서 구조를 `doc_service` 중심으로 통합하고, 벡터 임베딩 및 저장 과정을 분리하여 확장성 있는 문서 처리 파이프라인을 완성함
9	LLM 응답 시간 및 context 길이 로깅	**time, FastAPI logger**	성능 모니터링을 위한 처리 시간 측정 및 context 길이 기록
10	대화 요약 기능 추가	**KoBART, KoGPT, Transformers**	너무 긴 context를 줄이기 위해 대화 내용을 요약하는 기능 추가
11	검색 결과 chunk 하이라이트 또는 로깅	**ChromaDB, FastAPI**	RAG가 어떤 chunk를 검색에 사용했는지 시각화하거나 로그에 남김
12	간단한 인증 또는 사용자별 문서 분리	**API Key, JWT, 사용자 ID 처리**	사용자 인증을 통해 데이터 분리 및 보안 강화

A. 예외 핸들러

1. 예외 핸들러 별도 파일로 관리 : `app/exceptions/exception_handlers.py`

- FastAPI 프로젝트에서 모든 에러 응답을 일관된 JSON 포맷으로 처리

2. `main.py` 에서 등록

```
from fastapi import FastAPI
from fastapi.exceptions import RequestValidationError
from starlette.exceptions import HTTPException as StarletteHTTPException
from app.exceptions import exception_handlers  # 추가

app = FastAPI()
```

```
# 예외 핸들러 등록
app.add_exception_handler(Exception, exception_handlers.general_exception_handler)
app.add_exception_handler(StarletteHTTPException,
exception_handlers.http_exception_handler)
app.add_exception_handler(RequestValidationError,
exception_handlers.validation_exception_handler)
```

3. 테스트

```
# 존재하지 않는 경로 → 404
curl http://localhost:48001/invalid-path

# 필수 파라미터 빠짐 → 422
curl http://localhost:48001/sessions

# 일부러 서버 오류 유도
# (예: 세션 조회에 user_id 빼먹고 내부 코드에서 None 접근하도록 수정해 테스트)
```

---

6. 세션 조회/삭제/초기화 기능

1) API 명세

설명	메서드	URL
세션 목록 조회	`GET`	`/sessions?user_id=user123`
단일 세션 상세 조회	`GET`	`/sessions/{session_id}?user_id=user123`
세션 삭제	`DELETE`	`/session/{session_id}`
전체 세션 삭제	`DELETE`	`/sessions?user_id=user123`

2) 소스코드 작성

1. `session_store.py` 확장 코드

- delete_session() 추가
- clear_all_sessions() 추가

2. 라우터

- `session_router.py` : FastAPI 라우터 구현
- `main.py` : session_router 등록

3) 테스트

```
# 1. 세션 목록 조회
curl "http://localhost:48001/sessions?user_id=user123"

# 2. 단일 세션 조회
curl "http://localhost:48001/sessions/sess001?user_id=user123"

# 3. 단일 세션 삭제
curl -X DELETE "http://localhost:48001/sessions/sess001?user_id=user123"

# 4. 전체 세션 삭제
curl -X DELETE "http://localhost:48001/sessions?user_id=user123"
```

---

7. 업로드 문서 관리 기능

1) API 설계

설명	메서드	URL
업로드한 문서 목록 조회	`GET`	`/documents`
특정 문서 삭제	`DELETE`	`/documents/{source}`

2) 소스코드 작성

1. 서비스 분리

- `embed_service.py` : 순수 기능 유틸성 함수 (임베딩 및 청크 분할)
- `doc_service.py` : 문서 저장/관리/삭제 처리자

2. 라우터

- `doc_router.py` : FastAPI 라우터 구현

3) 테스트

```
# 파일 업로드
curl -X POST http://localhost:48001/documents/upload-doc \
  -H "Content-Type: multipart/form-data" \
  -F "file=@/labs/docker/images/chat-dev-sjchoi/src/volumns/fastapi/test_doc.txt"

# 업로드된 문서 목록 조회
curl -X GET http://localhost:48001/documents

# 문서 삭제 요청
# 정상 요청 (성공) : uuid 맞게 변경
curl -X DELETE "http://localhost:48001/documents/test.txt" \
  -H "Content-Type: application/json" \
  -d '{"uuid": "279910e8-07e8-4b6c-8b53-d7a120da5b5e"}'
# 오류 요청 (uuid 누락)
```

```
curl -X DELETE "http://localhost:48001/documents/test.txt" \
  -H "Content-Type: application/json" \
  -d '{}'
```

B. 세션 및 챗 서비스 구조 리팩터링

1) 세션 기능을 session_service 중심으로 분리

> 문제: 기존에는 session_router.py가 session_store.py를 직접 호출하여 Redis와 바로 통신 → 이는 로직 분리 원칙에 어긋나며, 향후 세션 저장소 교체(예: DB) 시 유연성이 떨어짐

1. `session_service.py` 생성 및 다음 기능을 위임:

- 세션 목록 조회 (get_user_sessions)
- 단일 세션 조회 (get_history)
- 단일 세션 삭제 (delete_session)
- 전체 세션 삭제 (clear_all_sessions)

2. `session_router.py` 수정

- session_store를 직접 import 하지 않고 무조건 session_service를 통해 접근하도록 변경

2) 챗 기능을 chat_service 중심으로 정리

> 문제: chat_with_session() 위치가 chat 서비스의 역할을 하면서도 session 중심 구조로 흩어져 있어 RAG 파이프라인 구조와 맞지 않음

1. `chat_with_session()` 메소드를 → `chat_service.py`로 이동
2. `chat_with_session()` 도 RAG 파이프라인을 따르게 변경

- chat_with_context() 구조처럼 `search_similar_docs()` → `build_prompt()` → `call_llm()` 흐름을 동일하게 적용

3) 테스트

```
# 세션 기반 챗 테스트 명령어
curl -X POST http://localhost:48001/chat-session \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": "user123",
    "session_id": "session001",
    "query": "한국의 수도는 어디인가요?"
}'
```

---

8. 문서 업로드 & 임베딩 구조 정리

> 다양한 형식의 문서(txt, pdf, docx, url)를 지원하여 RAG 기반 질의에 활용될 수 있도록 텍스트를 추출하고, 벡터로 임베딩하여 DB에 저장하는 흐름 구축

1. 책임 분리 구조

모듈	파일명	역할 및 책임
라우터	`doc_router.py`	- 업로드 API 정의 - 업로드된 파일 또는 URL을 받아 서비스에 전달
서비스	`doc_service.py`	- 입력 받은 파일/URL에서 텍스트 추출 **(파서 호출)** - 청크 분리 → 임베딩 → ChromaDB 저장
유틸	`parse_document.py`	- `txt`, `pdf`, `docx`, `url` 등 다양한 입력을 텍스트로 변환하는 기능만 담당

2. 처리 가능한 형식

유형	설명	파서 함수
`.txt`	일반 텍스트 파일	`extract_text_from_txt()`
`.pdf`	PDF 문서	`extract_text_from_pdf()`
`.docx`	Word 문서	`extract_text_from_docx()`
`url`	웹 URL 내용	`extract_text_from_url()`

3. 소스 코드 정리

- `src/volumns/fastapi/utils/parse_document.py` 생성
- `doc_service.py` 수정
- `doc_router.py` 수정

4. `requirements.txt` - 문서 파싱용 추가 필수 라이브러리 추가

- 파일 수정
- docker compose 재시작

5. 테스트

```
# SWAGGER-UI 확인 재기동 확인용
curl http://localhost:48001/docs

# 파일 업로드(.txt, .pdf, .docx) 테스트
curl -X POST "http://localhost:48001/documents/upload-doc" \
  -F "file=@/labs/docker/images/chat-dev-sjchoi/src/volumns/fastapi/test_doc.txt"

# URL 업로드 (웹 페이지 텍스트 크롤링)
curl -X POST "http://localhost:48001/documents/upload-doc?
url=https://en.wikipedia.org/wiki/Retrieval-augmented_generation"

# 파일 업로드 목록 확인
curl -X GET http://localhost:48001/documents
```