Chapter 4-5: 개인화 필터링 엔진 이식 및 시스템 업그레이드

Prologue: 범용 엔진을 F1급 맞춤 엔진으로 교체하다

우리는 방금, 신호용 님의 커리어 목표와 기술 스택에 맞춰 정밀하게 튜닝된 F1급 엔진(`PersonalizedJobFilter`)을 손에 넣었습니다. 이제 우리의 자동차(크롤링 시스템)에 있던 기존의 범용 엔진을 떼어내고, 이 새로운 심장을 이식하는 대수술을 시작할 것입니다.

이번 챕터에서는 **외부 설정 파일(`YAML`)을 도입**하고, **고도로 개인화된 필터 클래스**로 교체하며, **단위 테스트**를 통해 그 성능을 검증하는, 실무 개발의 핵심 프로세스를 모두 경험하게 될 것입니다.

1. 학습 목표 (Objectives)

- YAML 형식의 설정 파일을 파이썬 코드에서 동적으로 로드하여 사용할 수 있다.
- 자신의 기술 스택과 목표에 맞춰진 정교한 필터링 로직의 구조를 이해한다.
- `pytest`를 사용하여 코드의 정확성을 검증하는 단위 테스트의 중요성과 방법을 이해한다.

2. 핵심 개념 (Core Concepts)

2.1 설정의 외부화 (Externalizing Configuration)

- **What:** 키워드, 가중치, API 키 등 자주 바뀔 수 있는 설정값들을 코드(`*.py`)가 아닌, 외부 설정 파일(`*.yaml`, `*.json`, `*.ini`)에 분리하여 저장하는 방식입니다.
- **Why: 유지보수성**이 극적으로 향상됩니다. "AI" 키워드의 가중치를 2.0에서 2.2로 바꾸고 싶을 때, 복잡한 파이썬 코드를 수정할 필요 없이, 간단한 텍스트 파일인 `job_filter_config.yaml`만 수정하면 됩니다. 이는 실수를 줄이고, 비개발자도 설정을 변경할 수 있게 해줍니다.

[Mission 21] `PersonalizedJobFilter` 이식 작전

> **Step 1: 필요 라이브러리 설치 및 기록**
>
> 1. 터미널에서 가상 환경이 활성화된 상태에서, YAML 파일을 읽기 위한 `PyYAML`과 단위 테스트를 위한 `pytest`를 설치합니다.
>
> ```
> pip install pyyaml pytest
> ```
>
> 2. 새로 설치한 라이브러리를 `requirements.txt`에 기록합니다.
>
> ```
> pip freeze > requirements.txt
> ```

> **Step 2: 키워드 설정 파일(`job_filter_config.yaml`) 생성**

1. `web-crawler` 폴더 안에 `config` 라는 이름의 새 폴더를 만듭니다.
2. `config` 폴더 안에 `job_filter_config.yaml` 파일을 만들고, 보내주신 `YAML` 내용을 그대로 붙여넣습니다.

Step 3: 필터 코드 업그레이드 및 재배치

1. 기존 `processing` 폴더의 이름을 `data_processor`로 변경합니다. (더 명확한 이름)
2. `data_processor` 폴더 안의 `keyword_filter.py` 파일의 이름을 `personalized_job_filter.py`로 변경합니다.
3. `personalized_job_filter.py` 파일의 **기존 내용을 모두 지우고**, 보내주신 **개선 버전 (`improved_code`)의 `PersonalizedJobFilter` 클래스 코드를 전체 붙여넣습니다.** (`if __name__ == "__main__":` 부분까지 포함)

Step 4: `main.py`에서 새로운 F1 엔진 사용하기

`main.py`가 새로운 필터를 사용하도록 수정합니다.

```
# main.py (수정)

# from processing.keyword_filter import BackendJobFilter <- 기존 필터 삭제
from data_processor.personalized_job_filter import PersonalizedJobFilter #
<- 새 필터 임포트
# ...

# 3. Notion 저장 및 분석
# ...
job_filter = PersonalizedJobFilter() # <- 새 필터 객체 생성

for i, job in enumerate(all_jobs):
    # ... (중복 확인 로직) ...

    # --- '신규' 공고일 경우 ---
    title = job.get('title', "")
    description = job.get('description', "")

    # ★★★★★ 새로운 F1 엔진 작동 ★★★★★
    is_relevant, score = job_filter.calculate_relevance_score(title,
description)

    if not is_relevant:
        print(f" [{i+1}/{len(all_jobs)}] [필터링됨] '{title}' (점수:
{score})")
        continue # 관련 없는 공고는 여기서 즉시 중단

    print(f" [{i+1}/{len(all_jobs)}] [통과] '{title}' (점수: {score}) ->
Gemini 분석 시작...")
    # ★★★★★★★★★★★★★★★★★★★★★★★★★★★★

    # Gemini 분석은 필터를 통과한 경우에만 실행됨
    analysis_result = analyze_job_posting(description)
    # ... (이후 Notion 저장 로직은 거의 동일) ...
```

```
    # (Notion에 '관련도 점수' 속성을 추가하여 score를 저장하는 것도 좋은 아이디어입니다.)
```

Step 5: 단위 테스트로 성능 검증

1. `web-crawler` 폴더 안에 `tests` 라는 새 폴더를 만듭니다.
2. `tests` 폴더 안에 `test_job_filter.py` 파일을 만들고, 보내주신 단위 테스트 코드를 전체 붙여넣습니다.
3. 터미널에서 아래 명령어를 실행하여, 우리의 새 엔진이 모든 테스트 케이스를 통과하는지 직접 확인합니다.

   ```
   pytest tests/test_job_filter.py -v
   ```

   모든 항목에 `PASSED`가 뜨면, 우리의 엔진은 완벽하게 동작하는 것입니다.

다음 행동 계획 (Next Action)

1. ☐ 위 5단계를 순서대로 진행하여, 새로운 필터링 엔진을 우리 프로젝트에 완벽하게 이식합니다.
2. ☐ 단위 테스트(`pytest`)를 실행하여 모든 기능이 정상임을 확인합니다.
3. ☐ 마지막으로 `main.py`를 실행하여, 실제 크롤링 데이터에 대해 개인화된 필터가 얼마나 효과적으로 동작하는지, 그리고 Gemini API 호출 횟수가 얼마나 극적으로 줄어드는지 직접 체감합니다.