Docker 외부와 내부 간 파일 복사를 필요에 따라 수동으로 수행하는 방식은 깔끔하고 명확하며, 작업에 대한 제어력을 유지할 수 있는 좋은 선택입니다.

아래에 외부와 내부 간 파일 복사를 위한 구체적인 명령과 실무적인 팁을 정리했습니다.

**1. Docker 외부 → 내부 파일 복사**

**명령어:**

docker cp <외부 파일 경로> <컨테이너 이름 또는 ID>:<내부 경로>

**예시:**

docker cp /home/dongwank/dev/python/NaverBlogCrawler/NaverBlogCrawler/spiders/ naver\_blog\_spider.py naverblogcrawler-scrapy-1:/app/NaverBlogCrawler/spiders

**결과:**

* naver\_blog\_spider.py가 Docker 컨테이너 naverblogcrawler-scrapy-1의 /app 디렉터리에 복사됩니다.

**2. Docker 내부 → 외부 파일 복사**

**명령어:**

docker cp <컨테이너 이름 또는 ID>:<내부 파일 경로> <외부 경로>

**예시:**

docker cp naverblogcrawler-scrapy-1:/app/naver\_blog\_2024\_11\_27.json /home/dongwank/dev/python/NaverBlogCrawler

**결과:**

* naver\_blog\_2024\_11\_27.json 파일이 Docker 컨테이너 /app 디렉터리에서 호스트 머신의 /home/dongwank/dev/python/NaverBlogCrawler 디렉터리로 복사됩니다.

**3. 전체 디렉터리 복사**

**외부 → 내부 디렉터리 복사:**

docker cp /path/to/local/directory <컨테이너 이름 또는 ID>:<내부 경로>

**내부 → 외부 디렉터리 복사:**

docker cp <컨테이너 이름 또는 ID>:<내부 디렉터리 경로> /path/to/local

**예시:**

* 외부 디렉터리 /home/dongwank/dev/python/NaverBlogCrawler/를 내부 /app/에 복사:

docker cp /home/dongwank/dev/python/NaverBlogCrawler naverblogcrawler-scrapy-1:/app

* 내부 디렉터리 /app/를 외부 /home/dongwank/dev/python/NaverBlogCrawler/로 복사:

docker cp naverblogcrawler-scrapy-1:/app /home/dongwank/dev/python/NaverBlogCrawler

**4. 파일 복사 시 주의점**

1. **경로 끝 슬래시(/) 주의**:
   * 경로 끝에 슬래시(/)를 붙이면 복사 동작이 달라질 수 있습니다.
     + 슬래시 있음: 디렉터리 내용만 복사
     + 슬래시 없음: 디렉터리 자체를 복사
2. **복사 대상 경로 존재 여부**:
   * 복사하려는 경로가 컨테이너 내부에 이미 존재해야 합니다. 없는 경우 복사 시 실패하거나 자동으로 생성되지 않을 수 있습니다.
3. **권한 문제**:
   * Docker 컨테이너 내부에서는 root 권한으로 파일이 복사됩니다. 권한 문제를 방지하려면 복사 후 적절한 권한 설정이 필요할 수 있습니다:

chmod 755 /path/to/file

**5. 실무 팁**

1. **파일 복사 자동화**:
   * 반복적인 파일 복사가 필요하다면, 간단한 스크립트를 작성하여 작업을 자동화하세요.
2. **컨테이너 내부 파일 관리**:
   * Docker 내부에서 수정된 파일을 외부로 복사하지 않고 컨테이너를 삭제하면 데이터 손실이 발생합니다. 작업 종료 전에 필요한 파일은 항상 외부로 백업하세요.
3. **컨테이너 재사용 전략**:
   * 컨테이너를 멈춘 상태에서도 파일 복사가 가능하므로, 재사용 가능한 컨테이너를 유지하고 필요 시 파일을 복사하는 방식으로 활용하면 효율적입니다.

**6. 명령 예제 스크립트**

**외부에서 컨테이너로 파일 복사:**

#!/bin/bash

docker cp /path/to/local/file.py naverblogcrawler-scrapy-1:/app

**컨테이너에서 외부로 파일 복사:**

#!/bin/bash

docker cp naverblogcrawler-scrapy-1:/app/output.json /path/to/local

스크립트를 사용하면 반복 작업을 자동화할 수 있습니다.

필요한 작업이나 추가 설정이 있다면 언제든 문의해주세요.

효율적인 작업 환경 구축을 도와드리겠습니다!

 기본적으로 Docker 명령은 Docker 외부에서만 작동합니다.

 Docker 내부에서도 Docker CLI를 설정하고 Docker 소켓을 공유하면 Docker 명령을 실행할 수 있지만, 이는 일반적인 상황에서는 필요하지 않습니다. 보안 위험과 퍼포먼스 상 제약 사항이 있습니다.

Docker 컨테이너와 외부 환경을 완전히 독립적으로 설정하고, 필요 시에만 파일을 수동으로 복사하는 방식은 매우 깔끔하고 제어하기 쉬운 설정입니다. 이를 구현하는 단계와 주요 명령은 아래와 같습니다.

**1. Docker 컨테이너 생성**

외부 디렉토리와의 연결(볼륨 마운트) 없이 Docker 컨테이너를 생성하고 실행합니다.

**명령어:**

docker run -it --name naverblogcrawler naverblogcrawler-scrapy /bin/bash

**설명:**

* -it: 터미널 상호작용 모드로 실행.
* --name naverblogcrawler: 컨테이너 이름을 naverblogcrawler로 지정.
* naverblogcrawler-scrapy: 사용할 이미지 이름.
* /bin/bash: 컨테이너 내부에서 Bash 셸을 실행.

**2. 파일 복사: Docker 외부 ↔ 내부**

**2.1 외부에서 컨테이너 내부로 파일 복사**

필요한 파일을 Docker 외부에서 컨테이너 내부로 복사합니다.

**명령어:**

docker cp /path/to/local/file <컨테이너 이름>:<컨테이너 내부 경로>

**예시:**

docker cp /home/user/project/script.py naverblogcrawler:/app/script.py

**결과:**

* /home/user/project/script.py 파일이 컨테이너 내부 /app/script.py로 복사됩니다.

**2.2 컨테이너 내부에서 외부로 파일 복사**

컨테이너 내부 파일을 Docker 외부로 복사합니다.

**명령어:**

docker cp <컨테이너 이름>:<컨테이너 내부 파일 경로> /path/to/local

**예시:**

docker cp naverblogcrawler:/app/output.json /home/user/project/output.json

**결과:**

* 컨테이너 내부의 /app/output.json 파일이 외부 경로 /home/user/project/output.json로 복사됩니다.

**3. 파일 복사 없이 작업 독립적으로 유지**

컨테이너 내부와 외부를 완전히 독립적으로 설정하여 파일 복사를 하지 않는 경우, 아래처럼 각각의 환경에서 작업을 진행할 수 있습니다.

**컨테이너 내부에서 작업**

1. 컨테이너 내부로 진입:

docker exec -it naverblogcrawler /bin/bash

1. 컨테이너 내부 파일 작업:

cd /app

python script.py

**컨테이너 외부에서 작업**

외부에서는 외부 파일 시스템에서만 작업하며, 컨테이너에 영향을 주지 않습니다.

**4. 필요 없는 컨테이너 삭제**

더 이상 필요하지 않은 컨테이너는 삭제하여 환경을 깔끔하게 유지합니다.

**명령어:**

docker rm naverblogcrawler

**5. Docker 환경 정리**

컨테이너와 이미지 모두 삭제하여 환경을 완전히 초기화하려면 아래 명령어를 사용합니다.

**모든 컨테이너 삭제:**

docker rm $(docker ps -aq)

**모든 이미지 삭제:**

docker rmi $(docker images -q)

**정리**

* 컨테이너와 외부 환경을 독립적으로 유지하면서 필요한 경우에만 파일을 복사하려면 docker cp 명령을 사용하면 됩니다.
* 컨테이너 생성 시에는 볼륨 마운트 옵션을 사용하지 않습니다.
* 작업이 끝난 후에는 필요 없는 컨테이너와 이미지를 정리하여 환경을 간소화합니다.

docker ps -a 명령은 Docker에서 모든 컨테이너(실행 중이거나 종료된 컨테이너 포함)의 상태를 보여줍니다. 출력 항목의 각 필드에 대한 설명은 아래와 같습니다.

**1. CONTAINER ID**

* **설명**:
  + 컨테이너를 고유하게 식별하는 **16자리의 고유 ID**입니다.
  + ID의 앞부분(12자리)이 출력되며, 명령 실행 시 이를 통해 컨테이너를 식별하고 작업할 수 있습니다.
* **예시**:

CONTAINER ID

b7815fe2c755

* **활용**:
  + 특정 컨테이너를 대상으로 명령 실행:

docker stop b7815fe2c755

**2. IMAGE**

* **설명**:
  + 컨테이너가 생성된 Docker 이미지의 이름입니다.
  + 이 이미지는 컨테이너의 실행 환경(애플리케이션, 라이브러리, OS 등)을 정의합니다.
* **예시**:

IMAGE

naverblogcrawler-scrapy

* **활용**:
  + 사용된 이미지를 기준으로 새 컨테이너 생성:

docker run -it naverblogcrawler-scrapy /bin/bash

**3. COMMAND**

* **설명**:
  + 컨테이너 시작 시 실행된 기본 명령입니다.
  + Dockerfile의 CMD나 ENTRYPOINT에서 정의된 명령이 표시됩니다.
  + 명령의 전체 내용은 일부만 출력되며, 생략된 부분은 docker inspect로 확인 가능합니다.
* **예시**:

COMMAND

"/bin/bash"

* **활용**:
  + 컨테이너의 기본 명령을 확인해 실행 중 어떤 작업이 수행되는지 파악.

**4. CREATED**

* **설명**:
  + 컨테이너가 **언제 생성**되었는지 보여줍니다.
  + 생성된 시간과 경과 시간을 표시합니다.
* **예시**:

CREATED

24 hours ago

* **활용**:
  + 오래된 컨테이너를 확인하여 불필요한 컨테이너를 삭제:

docker rm <CONTAINER ID>

**5. STATUS**

* **설명**:
  + 컨테이너의 **현재 상태**를 표시합니다.
  + 주요 상태:
    - Up: 컨테이너가 실행 중.
    - Exited (code): 컨테이너가 종료됨 (종료 코드를 포함).
    - Paused: 컨테이너가 일시 중지 상태.
* **예시**:

STATUS

Exited (1) 10 seconds ago

* **활용**:
  + 상태를 기준으로 실행 중인 컨테이너만 확인:

docker ps

**6. PORTS**

* **설명**:
  + 컨테이너에서 외부로 노출된 네트워크 포트를 표시합니다.
  + Dockerfile의 EXPOSE 명령이나 docker run의 -p 옵션으로 설정한 포트 매핑 정보를 보여줍니다.
  + 형식:

<호스트 포트>:<컨테이너 포트>

* **예시**:

PORTS

8080->80/tcp

* + 호스트의 8080 포트가 컨테이너의 80 포트와 연결되어 있습니다.
* **활용**:
  + 포트 매핑을 기준으로 컨테이너의 네트워크 설정 확인.

**7. NAMES**

* **설명**:
  + 컨테이너의 **사용자 정의 이름** 또는 Docker가 자동 생성한 이름을 표시합니다.
  + 컨테이너 생성 시 --name 옵션으로 이름을 지정하거나, 자동으로 할당된 이름이 표시됩니다.
* **예시**:

NAMES

naverblogcrawler

* **활용**:
  + 이름으로 컨테이너를 제어:

docker start naverblogcrawler

docker exec -it naverblogcrawler /bin/bash

**정리**

| **필드** | **설명** |
| --- | --- |
| CONTAINER ID | 컨테이너를 고유하게 식별하는 ID (12자리). |
| IMAGE | 컨테이너가 생성된 Docker 이미지 이름. |
| COMMAND | 컨테이너 시작 시 실행된 기본 명령. |
| CREATED | 컨테이너가 생성된 시간. |
| STATUS | 컨테이너의 현재 상태 (Up, Exited, Paused). |
| PORTS | 컨테이너의 네트워크 포트 매핑 정보. |
| NAMES | 컨테이너의 사용자 정의 이름 또는 Docker 자동 생성 이름. |

이 정보를 활용해 컨테이너를 효율적으로 관리할 수 있습니다. 추가로 궁금한 점이 있으면 알려주세요!

Scrapy를 기본으로 하고, 필요할 경우 Selenium을 추가로 사용하는

방법을 선택하셨다면, Naver 블로그의 **HTML 구조 확인**은 중요한 첫 단계입니다. HTML 구조를 확인하면서 **Scrapy만으로 가능한 경우**와 **Selenium이 필요한 경우**를 구분할 수 있습니다.

아래는 HTML 구조 확인을 단계적으로 수행하는 방법입니다.

**1. HTML 구조 확인 목표**

* **필요한 데이터 위치 파악**: 게시글 제목, 내용, 작성자, 날짜 등이 HTML에서 어디에 위치하는지 확인.
* **동적 콘텐츠 판단**: 데이터가 HTML에 바로 있는지, 아니면 JavaScript로 동적으로 로드되는지 판단.
* **CSS Selector 작성**: Scrapy 또는 Selenium에서 사용할 정확한 CSS Selector를 작성.

**2. HTML 구조 확인 단계**

**2-1. 대상 URL 열기**

1. **크롬 브라우저에서 URL 열기**:
   * 키워드 검색 결과 URL 예:

https://section.blog.naver.com/Search/Post.naver?keyword=대한항공

* + 특정 블로그 URL 예:

https://blog.naver.com/navigator

1. **크롬 개발자 도구(F12) 열기**:
   * 브라우저에서 대상 URL로 이동한 뒤, \*\*개발자 도구(F12)\*\*를 열고 데이터가 포함된 HTML 구조를 확인합니다.

https://section.blog.naver.com/Search/Post.naver?keyword=대한항공

[검색결과]

글: **대한항공**에 대한 검색결과 입니다. **1,133,297건**

**블로그: 대한항공**에 대한 검색결과 입니다. **906건**

HTML 구조를 확인하기 위해 이동해야 하는 URL은 아래 두 가지 경우에 따라 다릅니다.

**1. 키워드 검색 결과 페이지 확인**

https://section.blog.naver.com/Search/Post.naver?keyword=대한항공

* **대상 URL**: 키워드 검색 결과 페이지 전체입니다.
* **목적**: 검색 결과 목록에서 개별 게시글의 링크를 추출하고, 각 링크로 이동해 데이터를 수집할 수 있도록 HTML 구조를 확인합니다.

**확인 방법:**

1. 브라우저에서 위 URL을 열어 **검색 결과 목록 페이지**를 확인합니다.
2. 크롬 개발자 도구(F12)를 열고 **검색 결과 목록의 HTML 구조**를 확인합니다.
   * 예: 게시글 제목, 작성자, 링크 등이 포함된 요소의 Selector 확인.
3. 각 게시글로 이동하기 위해 필요한 링크 정보(예: <a> 태그의 href 속성)를 파악합니다.

**2. 개별 게시글 페이지 확인**

검색 결과 목록에서 **하나의 게시글을 선택**하여 해당 URL로 이동합니다.

* **대상 URL**: 검색 결과 중 하나의 게시글 페이지.  
  예: https://blog.naver.com/navigator/123456789
* **목적**: 개별 게시글 페이지에서 데이터를 추출하기 위한 HTML 구조를 확인합니다.

**확인 방법:**

1. 검색 결과 목록에서 게시글 링크를 하나 클릭하여 게시글 페이지로 이동합니다.
2. 크롬 개발자 도구(F12)를 열고, **게시글 내용, 작성자, 날짜 등의 HTML 구조**를 확인합니다.
   * 예: 제목이 <h3 class="se-title-text"> 태그에 있고, 본문이 <div class="se-main-container"> 태그에 있을 수 있습니다.
3. 개별 게시글 데이터를 수집하기 위한 CSS Selector를 작성합니다.

**정리**

* **https://section.blog.naver.com/Search/Post.naver?keyword=대한항공**
  + **사용 목적**: 검색 결과 목록 페이지에서 게시글 링크를 크롤링하기 위해 HTML 구조를 확인.
  + Scrapy가 요청할 첫 번째 URL.
* **검색 결과 중 하나의 개별 게시글 URL (예: https://blog.naver.com/navigator/123456789)**
  + **사용 목적**: 게시글 본문, 작성자, 날짜 등 세부 데이터를 수집하기 위해 HTML 구조를 확인.
  + Scrapy가 크롤링할 게시글 데이터의 최종 URL.

**크롤링 흐름**

1. **검색 결과 페이지 크롤링**:  
   https://section.blog.naver.com/Search/Post.naver?keyword=대한항공에서 게시글 링크를 추출.
2. **개별 게시글 페이지 크롤링**:  
   추출한 링크를 따라가 각 게시글 페이지에서 데이터를 수집.

**추가 도움:**

필요하다면 각 단계에서 HTML 구조 확인 방법을 더 자세히 설명하거나 크롤링 코드로 바로 작성할 수 있도록 도와드릴게요.

**2-2. 데이터가 동적으로 로드되는지 확인**

1. **HTML 구조 확인**:
   * 개발자 도구에서 HTML을 열어 **게시글 목록** 또는 **게시글 내용**이 HTML에 포함되어 있는지 확인합니다.
     + HTML에서 데이터를 확인하려면 **Elements** 탭에서 원하는 데이터(예: 제목)를 검색합니다.
     + 개발자 도구(F12) > **Ctrl+F**를 눌러 키워드 검색.
     + 키워드 예: "대한항공", 게시글 제목, 작성자 등.
2. **네트워크 요청 확인**:
   * **Network 탭**으로 이동한 뒤, 페이지를 새로고침(F5)합니다.
   * **XHR** 필터를 사용해 JavaScript로 데이터를 가져오는 요청이 있는지 확인합니다.
     + 예: 특정 API 호출로 게시글 데이터를 가져오는 경우.
3. **동적 로딩 여부 판단**:
   * HTML에서 원하는 데이터가 바로 보이면 **Scrapy만으로 가능**.
   * HTML에서 데이터가 보이지 않고, Network 탭에서 데이터가 로드되는 것을 확인하면 **Selenium 필요**.

**2-3. CSS Selector 작성**

1. **HTML 요소 선택**:
   * 게시글 목록, 제목, 작성자, 날짜 등의 요소를 찾습니다.
   * 요소를 우클릭 → \*\*Inspect(검사)\*\*를 클릭합니다.
2. **CSS Selector 복사**:
   * 요소를 마우스 오른쪽 클릭 → **Copy → Copy selector**를 선택해 CSS Selector를 복사합니다.
   * 복사한 Selector가 올바른지 확인하려면 Console 탭에서 테스트:

document.querySelectorAll('<복사한 Selector>')

1. **Selector 확인 및 최적화**:
   * 복사한 Selector가 너무 길거나 불필요한 부분이 많다면 최적화합니다.
     + 예: div.post-container > ul > li.post-item > a.title-link
     + Selector를 간단하게 줄여서 원하는 데이터만 선택 가능하도록 수정.

**3. Scrapy와 Selenium 사용 여부 판단**

1. **Scrapy만으로 가능한 경우**:
   * HTML에 데이터가 그대로 포함되어 있고, JavaScript 렌더링 없이 접근 가능하다면 Scrapy만으로 충분합니다.
   * 작성한 CSS Selector를 Scrapy의 Spider 코드에 추가합니다.
2. **Selenium이 필요한 경우**:
   * 데이터가 JavaScript로 동적으로 로드되거나, XHR 요청으로 데이터를 받아오는 경우 Selenium을 추가로 사용해야 합니다.
   * 예를 들어, 게시글 목록이 동적으로 로드되는 경우 Selenium으로 로드된 HTML을 Scrapy와 함께 처리합니다.

**4. 실습 예시**

**Scrapy로 HTML 데이터 수집**

게시글 목록에서 제목, 링크, 작성자를 수집하는 예:

import scrapy

class NaverBlogSpider(scrapy.Spider):

name = "naver\_blog"

start\_urls = ["https://section.blog.naver.com/Search/Post.naver?keyword=대한항공"]

def parse(self, response):

for post in response.css("div.list\_search\_post > ul > li"):

title = post.css("a.desc\_inner::text").get()

link = post.css("a.desc\_inner::attr(href)").get()

author = post.css("span.name\_author::text").get()

yield {

"title": title,

"link": link,

"author": author,

}

**Selenium과 Scrapy 통합**

Selenium으로 JavaScript로 로드된 데이터를 처리한 뒤, Scrapy로 데이터를 파싱하는 예:

from scrapy\_selenium import SeleniumRequest

class NaverBlogSpider(scrapy.Spider):

name = "naver\_blog"

start\_urls = ["https://section.blog.naver.com/Search/Post.naver?keyword=대한항공"]

def start\_requests(self):

for url in self.start\_urls:

yield SeleniumRequest(url=url, callback=self.parse)

def parse(self, response):

for post in response.css("div.list\_search\_post > ul > li"):

title = post.css("a.desc\_inner::text").get()

link = post.css("a.desc\_inner::attr(href)").get()

author = post.css("span.name\_author::text").get()

yield {

"title": title,

"link": link,

"author": author,

}

**5. 최종 점검**

1. HTML에서 데이터가 올바르게 파싱되는지 테스트합니다.
2. Scrapy만으로 충분한지, Selenium이 필요한지 여부를 확인합니다.
3. 크롤링 대상 데이터와 HTML 구조가 계속 바뀌는지 모니터링하며 CSS Selector를 업데이트합니다.

추가로 도움이 필요하거나 작업 중 문제가 발생하면 알려주세요!

Docker 이미지 빌드

docker build -t <이미지 이름>:<태그> .

docker build -t naverblogcrawler:latest .

빌드 결과 확인

docker images

컨테이너 생성 및 실행

docker run -it --name <컨테이너 이름> <이미지 이름>:<태그>

docker run -it --name naverblogcrawler naverblogcrawler:latest

실행 로그 확인

docker logs <컨테이너 이름>

docker logs naverblogcrawler

중단된 컨테이너 재사용:

이미 생성된 컨테이너를 다시 실행하려면:

docker start -ai <컨테이너 이름>

내부 파일 확인:

docker exec -it <컨테이너 이름> /bin/bash

컨테이너 작업이 완료되면 데이터를 백업하거나 종료 후 삭제합니다:

docker stop <컨테이너 이름>

docker rm <컨테이너 이름>

1. docker start -ai <컨테이너 이름>

옵션 설명

-a (attach):

컨테이너의 표준 출력(stdout)과 표준 에러(stderr)에 연결합니다.

이 옵션을 사용하면 컨테이너 내부에서 실행된 명령의 출력 결과를 터미널에서 실시간으로 확인할 수 있습니다.

-i (interactive):

컨테이너의 표준 입력(stdin)에 연결하여 입력을 받을 수 있도록 합니다.

2. docker exec -it <컨테이너 이름> /bin/bash

옵션 설명

-i (interactive):

컨테이너의 표준 입력(stdin)에 연결하여 사용자 입력을 받을 수 있도록 합니다.

예: 컨테이너 내부에서 입력을 받아야 하는 Bash 셸 실행.

-t (tty):

컨테이너 내부에서 가상 터미널(pseudo-TTY)을 활성화합니다.

이 옵션은 터미널과 같은 인터페이스를 제공하여 셸 환경에서 작업하기 편리하게 만듭니다.

**1. Dockerfile 명령 중 여러 번 작성 시 마지막 것만 적용되는 항목**

**항목과 특징**

1. **FROM**:
   * **여러 번 작성 가능하지만, 하나의 Dockerfile에는 처음 등장하는 FROM만 기준**으로 실행됩니다.
   * 여러 개의 FROM이 있을 경우, 각 FROM은 독립적인 빌드 단계(Multi-Stage Builds)를 시작합니다.
   * **규칙**:
     + 첫 번째 FROM은 반드시 Dockerfile의 첫 줄이어야 합니다.
     + 추가 FROM은 Multi-Stage Builds에서 사용됩니다.
2. **ENTRYPOINT**:
   * 여러 번 작성하면 **마지막에 정의된 항목만 유효**합니다.
   * 이전 ENTRYPOINT는 무시됩니다.
3. **CMD**:
   * 여러 번 작성하면 **마지막에 정의된 항목만 유효**합니다.
   * 이전 CMD는 무시됩니다.

**2. 작성 순서가 중요한 항목**

**FROM (맨 처음에 작성해야 함)**

* Dockerfile의 **첫 번째 명령어**여야 합니다.
* 이유:
  + FROM은 베이스 이미지를 정의하며, Dockerfile의 모든 빌드는 이 베이스 이미지 위에서 이루어집니다.
  + FROM 이전에 다른 명령이 오면 Docker는 오류를 반환합니다.

**LABEL (보통 FROM 바로 다음에 위치)**

* 위치는 자유롭지만, 일반적으로 FROM 바로 다음에 작성됩니다.
* 이유:
  + 작성자의 메타데이터를 포함하기 때문에 다른 명령 전에 작성하여 정보를 먼저 전달합니다.

**WORKDIR (파일 작업 명령 전에 위치)**

* 이후 명령(RUN, COPY, CMD)에서 사용될 **기본 작업 디렉터리**를 설정합니다.
* WORKDIR 이전에 파일 작업(COPY, RUN)을 시도하면 **절대 경로**로 지정해야 하므로 비효율적입니다.
* 예:

WORKDIR /app

COPY . .

**COPY (RUN이나 CMD 실행 전에 위치)**

* 복사한 파일을 사용해야 하는 명령(RUN, CMD)보다 앞에 위치해야 합니다.
* 예:

COPY requirements.txt .

RUN pip install -r requirements.txt

**RUN (COPY 이후)**

* 컨테이너 내부에서 명령을 실행하므로, 필요한 파일이 복사된 뒤에 위치해야 합니다.
* 예:

COPY requirements.txt .

RUN pip install -r requirements.txt

**ENV (필요한 위치에 자유롭게 작성)**

* 환경 변수를 설정하는 명령이므로, 필요한 위치에 자유롭게 작성할 수 있습니다.
* 보통 작업 중 환경 변수를 사용하기 전에 작성합니다.
* 예:

ENV APP\_ENV=production

RUN echo $APP\_ENV

**ENTRYPOINT와 CMD (맨 마지막에 위치)**

* 컨테이너가 실행될 때 작동하므로, Dockerfile의 **맨 마지막에 위치**해야 합니다.
* 이유:
  + ENTRYPOINT와 CMD는 컨테이너 실행 시의 동작을 정의합니다.
  + 다른 명령 실행 결과와 무관하게 항상 마지막에 실행됩니다.

**3. 항목별 작성 규칙 요약**

| **항목** | **여러 번 작성 시** | **작성 순서** |
| --- | --- | --- |
| **FROM** | 각 FROM은 독립적인 빌드 단계 | 첫 번째 줄에 작성. |
| **LABEL** | 이전 라벨과 병합됨 | FROM 바로 다음에 작성(선택 사항). |
| **WORKDIR** | 마지막 설정만 유효 | RUN, COPY 등 작업 명령 전에 작성. |
| **COPY** | 여러 번 작성 가능 | RUN, CMD 등 파일을 사용하는 명령보다 앞에 작성. |
| **RUN** | 여러 번 작성 가능 | 파일 작업(COPY) 이후에 작성. |
| **ENV** | 마지막 값만 적용 | 자유롭게 작성 가능하지만 사용 전에 설정. |
| **ENTRYPOINT** | 마지막 설정만 유효 | 컨테이너 실행과 관련된 설정이므로 Dockerfile 맨 마지막에 작성. |
| **CMD** | 마지막 설정만 유효 | ENTRYPOINT 이후에 작성 가능하지만 보통 함께 마지막에 작성. |

**4. 작성 순서 예시**

**Dockerfile 예제:**

# 베이스 이미지

FROM python:3.10-slim

# 작성자 정보

LABEL maintainer="user@example.com"

# 작업 디렉터리 설정

WORKDIR /app

# 파일 복사

COPY requirements.txt ./

RUN pip install --no-cache-dir -r requirements.txt

# 애플리케이션 파일 복사

COPY . .

# 환경 변수 설정

ENV APP\_ENV=production

# 기본 실행 명령

ENTRYPOINT ["python"]

CMD ["app.py"]

**5. 결론**

1. **여러 번 작성 시 마지막 항목만 유효한 것**:
   * ENTRYPOINT, CMD, ENV, WORKDIR.
2. **작성 순서가 중요한 것**:
   * **FROM**: 반드시 맨 처음.
   * **ENTRYPOINT와 CMD**: 항상 맨 마지막.
   * **WORKDIR, COPY, RUN**: 작업 흐름에 따라 작성 순서 조정.

이미 작성된 docker-compose.yml 파일이 있다면 이후 작업이 크게 달라지지 않으며, 기존 파일을 기준으로 **Compose 명령어를 사용해 컨테이너를 실행하고 관리**하면 됩니다.

하지만 몇 가지 세부 사항을 점검하고, 상황에 따라 추가 작업이 필요할 수 있습니다.

**1. 현재 docker-compose.yml 파일 분석**

services:

scrapy:

build: .

volumes:

- .:/app # 현재 디렉토리를 컨테이너의 /app에 연결

command: scrapy crawl naver\_blog\_spider

**내용 설명**

* **services.scrapy**:
  + 서비스 이름이 scrapy로 정의되어 있습니다.
* **build: .**:
  + 현재 디렉토리에 있는 Dockerfile을 사용하여 이미지를 빌드합니다.
* **volumes**:
  + 현재 디렉토리(.)를 컨테이너의 /app 경로에 마운트합니다.
  + 컨테이너 내부에서 /app 디렉토리와 로컬 디렉토리 간 동기화가 가능합니다.
* **command**:
  + 컨테이너가 실행될 때 scrapy crawl naver\_blog\_spider 명령이 실행됩니다.

**2. 이후 작업 순서**

**1) Docker Compose로 컨테이너 실행**

docker-compose.yml을 기반으로 컨테이너를 실행합니다:

docker-compose up --build

* **--build**: Dockerfile을 다시 빌드하여 이미지를 최신 상태로 만듭니다.
* 컨테이너가 실행되면 scrapy crawl naver\_blog\_spider 명령이 자동으로 실행됩니다.

**2) 실행 상태 확인**

docker-compose ps

출력 예:

Name Command State Ports

-----------------------------------------------------------------

scrapy\_1 scrapy crawl naver\_blog\_spider Up

* State가 Up이면 컨테이너가 정상적으로 실행 중입니다.

**3) 컨테이너 로그 확인**

컨테이너 로그를 확인하여 Scrapy 작업 결과를 모니터링합니다:

docker-compose logs -f

**4) 컨테이너 종료**

docker-compose down

* 실행 중인 컨테이너와 관련된 네트워크, 볼륨을 모두 정리합니다.

**3. 추가적으로 고려할 사항**

**1) 필요 시 명령 변경**

* 현재 설정된 command: scrapy crawl naver\_blog\_spider는 고정된 명령입니다.
* 다른 명령을 실행하려면 docker-compose.override.yml을 사용하거나 명령을 동적으로 변경할 수 있습니다:

docker-compose run scrapy scrapy list

**2) 디버깅 및 쉘 접속**

컨테이너 내부에서 디버깅이 필요하면 다음 명령으로 접속합니다:

docker exec -it <container\_name> bash

**3) 다른 서비스 추가**

* 필요에 따라 docker-compose.yml에 다른 서비스를 추가할 수 있습니다(예: 데이터베이스, Selenium 등).

**4. 작업 흐름 비교**

**기존 답변**

* docker-compose.yml 파일이 없다면, 새로 생성하고 필요한 설정을 추가하는 과정이 포함됨.

**현재 상황**

* 이미 docker-compose.yml이 있으므로, 다음과 같은 작업에만 집중하면 됩니다:
  + docker-compose up --build로 컨테이너 실행.
  + 실행 중인 컨테이너에서 Scrapy 작업 확인.
  + 필요에 따라 command 변경 또는 추가 서비스 설정.

**5. 최종 요약**

1. 현재 docker-compose.yml 파일을 기준으로 작업을 진행합니다.
2. 다음 명령어로 실행과 관리를 수행하세요:
   * 실행: docker-compose up --build
   * 로그 확인: docker-compose logs -f
   * 종료: docker-compose down
3. 필요 시 command를 수정하거나 추가 설정을 통해 작업을 확장할 수 있습니다.

작업 도중 **새로운 라이브러리가 필요**하다고 발견되면, 이를 설치하고 Docker 이미지와 컨테이너에 반영해야 합니다. 아래는 필요한 라이브러리를 설치하고 반영하는 방법입니다.

**1. 새로운 라이브러리 추가 및 반영**

**1.1 requirements.txt 파일 업데이트**

* 추가로 필요한 라이브러리를 requirements.txt 파일에 추가합니다.

echo "new-library-name" >> requirements.txt

* 예: pandas와 같은 라이브러리를 추가해야 한다면:

echo "pandas" >> requirements.txt

**1.2 Docker 이미지 다시 빌드**

docker-compose.yml 파일에 정의된 설정을 기반으로 이미지를 다시 빌드합니다:

docker-compose build

* 이 명령은 Dockerfile을 다시 실행하여 새로 추가된 라이브러리를 설치합니다.

**1.3 기존 컨테이너 종료 및 재실행**

* 실행 중인 컨테이너를 종료:

docker-compose down

* 새로 빌드된 이미지를 기반으로 컨테이너를 실행:

docker-compose up -d

**2. 작업 중 임시로 라이브러리 설치**

작업 도중 빠르게 테스트하기 위해 임시로 컨테이너 내부에 라이브러리를 설치할 수 있습니다. 다만, 이 방식은 컨테이너를 재시작하거나 이미지를 다시 빌드하면 설치된 라이브러리가 사라지므로, 최종적으로 requirements.txt에 반영해야 합니다.

**2.1 컨테이너 내부에서 임시 설치**

1. 실행 중인 컨테이너에 접속:

docker exec -it <container\_name> bash

1. 필요한 라이브러리 설치:

pip install new-library-name

1. 컨테이너 내부에서 테스트를 진행합니다.

**2.2 설치한 라이브러리를 requirements.txt에 반영**

컨테이너 내부에서 설치한 라이브러리를 requirements.txt에 추가합니다:

1. 설치된 라이브러리 확인:

pip freeze | grep new-library-name

1. 결과를 로컬 requirements.txt 파일에 추가:

pip freeze | grep new-library-name >> requirements.txt

**3. 새 라이브러리 설치 후 전체 프로세스**

1. **라이브러리 추가**
   * requirements.txt에 새 라이브러리 추가.
2. **Docker 이미지 재빌드**

docker-compose build

1. **컨테이너 재실행**

docker-compose down

docker-compose up -d

1. **컨테이너 내부 테스트**

docker exec -it <container\_name> bash

**4. 추가 팁**

**1) 자주 변경되는 경우**

* 작업 중 라이브러리를 자주 추가하는 경우, pip install로 테스트한 뒤 한 번에 requirements.txt에 반영하고 이미지를 다시 빌드하는 것이 효율적입니다.

**2) 라이브러리 버전 고정**

* 호환성 문제를 방지하기 위해 설치 시점의 버전을 고정하는 것이 좋습니다.
* requirements.txt에 명시된 형식:

library-name==1.2.3

**5. 요약**

1. 새로운 라이브러리가 필요하면:
   * requirements.txt에 추가.
   * Docker 이미지를 다시 빌드 (docker-compose build).
2. 빠르게 테스트하려면:
   * 컨테이너 내부에서 pip install로 설치.
   * 이후 requirements.txt에 반영하고 이미지를 재빌드.

**1. 기본 설정**

**1.1 프로젝트 초기 상태 확인**

1. 기존에 생성된 Docker 이미지와 컨테이너가 없는지 확인합니다:

docker-compose down --rmi all --volumes

* + **이미지 제거**: --rmi all
  + **볼륨 제거**: --volumes

1. 디렉토리 구조 확인: 프로젝트 디렉토리 구조가 아래와 같은지 확인합니다.

NaverBlogCrawler/

├── Dockerfile

├── docker-compose.yml

├── requirements.txt

├── scrapy.cfg

└── NaverBlogCrawler/

├── \_\_init\_\_.py

├── items.py

├── middlewares.py

├── pipelines.py

├── settings.py

└── spiders/

├── \_\_init\_\_.py

└── naver\_blog\_spider.py

**2. 개발 환경 설정**

개발 중에는 Python 코드 수정 및 필요한 라이브러리 설치 작업이 중심이 됩니다. 이를 위해 docker-compose.override.yml 파일을 생성하여 Docker Compose의 기본 설정을 확장하거나 대체합니다.

**2.1 docker-compose.override.yml 작성**

프로젝트 루트에 docker-compose.override.yml 파일을 생성하고 아래 내용을 작성합니다:

services:

scrapy:

command: bash # 기본 명령을 bash로 변경해 개발 환경 제공

stdin\_open: true # bash에서 작업 가능하도록 설정

tty: true # bash에서 작업 가능하도록 설정

* **command: bash**: 컨테이너가 실행되면 기본적으로 bash 환경에 들어가도록 설정.
* **stdin\_open, tty**: VS Code 및 Docker Compose 터미널에서 bash를 유지하기 위한 설정.

**3. 작업 흐름**

**3.1 초기 컨테이너 빌드 및 실행**

Docker Compose로 컨테이너를 빌드하고 실행합니다 (컨테이너 외부에서 실행):

docker-compose up -d

* **빌드 및 실행 과정**:
  + docker-compose.yml과 docker-compose.override.yml이 병합됩니다.
  + 컨테이너는 bash로 실행되며, Python 코드 수정 및 라이브러리 설치 작업이 가능합니다.

**3.2 Python 코드 수정**

1. **VS Code로 디렉토리 열기** (컨테이너 외부에서 실행):

code .

1. NaverBlogCrawler/spiders/naver\_blog\_spider.py 파일을 수정하여 Scrapy 코드를 업데이트합니다.

**3.3 필요한 라이브러리 설치**

1. **컨테이너 내부 접속**:

먼저 컨테이너 내부에 들어갑니다:  
컨테이너에 다시 들어가야 할 경우, 아래 명령을 사용합니다:

docker-compose exec scrapy bash

이 명령은 컨테이너를 새로 시작하지 않고, 이미 실행 중인 컨테이너에 접속합니다.

컨테이너 내부에서 Python 스크립트 실행, 추가 작업 등을 진행할 수 있습니다.

위 명령을 실행하면 컨테이너의 **bash 셸**에 접속합니다.

Prompt가 변경됩니다.

1. **라이브러리 설치** (컨테이너 내부에서 실행):

pip install <new-library>

1. **requirements.txt 업데이트** (컨테이너 내부에서 실행):   
   컨테이너 내부에서 설치한 라이브러리를 requirements.txt에 추가합니다:

pip freeze | grep <new-library> >> requirements.txt

1. **업데이트된 requirements.txt 적용 설치** (컨테이너 외부에서 실행):   
   라이브러리 설치 작업이 끝나면 exit 명령으로 컨테이너에서 빠져나옵니다:  
   exit  
   Prompt가 다시 WSL2 Ubuntu 환경으로 돌아옵니다.  
   컨테이너를 다시 빌드하여 업데이트된 라이브러리가 반영되도록 합니다:

docker-compose down

docker-compose up –build

**컨테이너 실행 상태 확인**:

docker-compose ps

**3.4 최종 명령 실행**

1. 개발 작업이 완료되면, **docker-compose.override.yml 파일을 삭제하거나 command를 주석 처리**합니다:

services:

scrapy:

# command: bash

1. 컨테이너를 재실행하여 scrapy crawl naver\_blog\_spider 명령을 실행합니다:

docker-compose up

**4. 추가 사항**

**4.1 컨테이너 Prompt 설정**

컨테이너 내부에서 **Prompt에 Docker Name과 경로 표시**를 유지하려면, 이미 Dockerfile에 포함된 다음 줄이 이를 지원합니다:

RUN echo 'export PS1="\\u@${CONTAINER\_NAME:-default-container}:\\w\\$ "' >> /etc/profile

Prompt는 다음과 같이 표시됩니다:

root@naver-blog-crawler:/app$

**4.2 Docker Compose로 불가능한 작업**

docker-compose.yml과 docker-compose.override.yml로 대부분의 작업이 가능하지만, 아래 작업은 Docker Compose 외의 명령이 필요할 수 있습니다:

1. **이미 실행 중인 컨테이너에 접근**:
   * docker exec 명령이 필요합니다:

docker-compose exec scrapy bash

1. **컨테이너 상태 디버깅**:
   * 특정 로그나 프로세스를 확인하려면 추가 명령 사용:

docker-compose logs

docker-compose ps

**5. 최종 요약**

1. **초기 작업**:
   * 기존 Docker 이미지와 컨테이너를 모두 제거:

docker-compose down --rmi all --volumes

* + docker-compose.yml과 docker-compose.override.yml 준비.

1. **개발 환경**:
   * docker-compose.override.yml에 command: bash 설정.
   * Python 코드 수정 및 라이브러리 추가 설치.
2. **최종 실행**:
   * docker-compose.override.yml의 command를 제거하거나 주석 처리.
   * scrapy crawl naver\_blog\_spider 실행.

이 작업 순서를 따르면 Docker Compose 중심으로 효율적인 작업을 진행할 수 있습니다.

docker-compose up -d

여기에서 -d 옵션은 Docker Compose에서 "detach mode"를 의미합니다.

즉, docker-compose up 명령을 실행한 후 터미널을 차단하지 않고 백그라운드에서 컨테이너를 실행합니다.

[동작 차이]

1. Without -d

컨테이너 로그가 현재 터미널에 출력됩니다.

터미널이 해당 프로세스를 차단합니다.

예:

docker-compose up

출력:

Creating network "example\_default" with the default driver

Creating example\_service ... done

Attaching to example\_service

example\_service | Starting service...

사용 사례:

디버깅이나 로그를 실시간으로 확인하고 싶을 때.

2. With -d

컨테이너가 백그라운드에서 실행됩니다.

터미널이 즉시 반환되어 다른 작업을 계속할 수 있습니다.

로그를 보려면 추가 명령이 필요합니다(예: docker logs 또는 docker-compose logs).

예:

docker-compose up -d

출력:

Creating network "example\_default" with the default driver

Creating example\_service ... done

추가 작업: 로그 확인

[사용 시기]

-d 사용:

컨테이너가 정상적으로 실행 중인지 확인만 하고, 이후 작업을 계속 진행하고 싶을 때.

로그가 필요하지 않거나 나중에 별도로 확인할 때.

-d 미사용:

컨테이너 실행 중 발생하는 문제를 실시간으로 확인하고 싶을 때.

로그를 바로 보고 디버깅할 때.