

포팅 매뉴얼

☰ 태그

개발환경

- 버전
 - 프로젝트 사용 도구
- 배포환경
 - Jenkins 설정 및 파이프라인
 - Jenkins 파이프라인
 - FastAPI 설정
- 환경변수 및 설정파일
 - API Server application.yml 및 application-product.yml
- Nginx 설정파일
- Airflow 설정파일
 - 1. Airflow docker image pull
 - 2. Dockerfile 생성
 - 3. Docker Image Build
 - 4. docker-compose.yml
 - 5. DAG 파일
- 외부 서비스
- DB 덤프 파일 최신본

개발환경

버전

💡 프로젝트에서 사용한 프로그램들의 버전을 정리합니다.

• Product Server

Java	17
Spring Boot	3.3.3
Gradle	gradle-8.8
Jenkins	2.478
Nginx	1.18.0
VS Code	1.91.1
IntelliJ	17.0.11+1-b1207.24 amd64
MySQL	8.0.39
MongoDB	7.0.14
Redis	7.4.0(docker)
Airflow	2.10.2(docker)
FastAPI	0.68.2(docker)

• Hadoop Server

Java	8
Python	3.8
Hadoop	3.3
Spark	3.5.3

프로젝트 사용 도구

💡 프로젝트 과정에서 사용한 도구들입니다.

이슈 관리	JIRA
형상 관리	Gitlab
커뮤니케이션	Notion, Mattermost
디자인	Figma
UCC	Movavi video editor
CI/CD	EC2, Jenkins, Docker

배포환경

💡 서버구성에 사용된 코드를 정리합니다.

Jenkins 설정 및 파이프라인

💡 Jenkins 도커 설정파일 및 CI/CD 파이프라인 스크립트입니다

Dockerfile 및 docker-compose.yml

```
# docker-compose.yml

services:
  app:
    image: openjdk:17-jdk-slim
    container_name: springboot-app
    volumes:
```

```

- ./backend/spring/build/libs/mutualrisk-0.0.1-SNAPSHOT.jar:/app.jar
ports:
- "8080:8080"
command: ["java", "-jar", "/app.jar"]
environment:
- SPRING_PROFILES_ACTIVE=prod
- TZ=Asia/Seoul
depends_on:
- redis
restart: always

redis:
image: "redis:alpine"
container_name: redis
ports:
- "6379:6379"
volumes:
- ./redis-data:/data
restart: always

```

Jenkins 파이프라인

Backend 배포 구성

▼ Build

```

pipeline{
  agent any

  stages{

    stage('shutdown old process') {
      steps {
        sh """
          #!/bin/bash
          ls -al

          # docker 내리기
          echo "Stop docker container..."
          docker compose down
        """
      }
    }

    stage('github clone (BE)') {
      steps {
        script {
          git branch: 'BE-Develop', credentialsId: 'gitlab_token',
            url: 'https://lab.ssafy.com/s11-bigdata-dist-sub1/S11P21A607.git'
        }
      }
    }

    stage('build (BE)') {
      steps {
        dir("./backend/spring") {
          sh """
            chmod +x ./gradlew
            ./gradlew clean build
          """
        }
      }
    }

    stage('restart docker') {
      steps {
        sh """
          # 다시 docker-compose 실행
          echo "Starting Docker Compose..."
          docker compose up -d

          echo "Docker Compose started successfully."
        """
      }
    }
  }

  post {
    always {
      script {
        def message
        if (currentBuild.result == 'SUCCESS') {
          message = "빌드 성공: ${env.JOB_NAME} #${env.BUILD_NUMBER} \n(<${env.BUILD_URL}|Details>)"
          mattermostSend (color: 'good',
            message: message,
            endpoint: 'https://meeting.ssafy.com/hooks/db9zfmhfbrjtgdmdb3ozbi8e9o',
            channel: 'jenkins_build'
          )
        } else if (currentBuild.result == 'FAILURE') {
          message = "빌드 실패: ${env.JOB_NAME} #${env.BUILD_NUMBER} \n(<${env.BUILD_URL}|Details>)\n"
          mattermostSend (color: 'danger',
            message: message,
            endpoint: 'https://meeting.ssafy.com/hooks/db9zfmhfbrjtgdmdb3ozbi8e9o',
            channel: 'jenkins_build'
          )
        } else {

```

```
message = "빌드 상태: ${env.JOB_NAME} #${env.BUILD_NUMBER} \n(<${env.BUILD_URL}|Details>)"
mattermostSend (
    message: message,
    endpoint: 'https://meeting.ssafy.com/hooks/db9zfmfhbrjtgdbmb3ozbi8e9o',
    channel: 'jenkins_build'
)
}
}
}
}
}
```

FrontEnd 배포 구성

▼ Build

```

pipeline{
  agent any

  tools {
    nodejs 'nodejs-20.15.0'
  }

  stages{

    stage('delete previous build') {
      steps {
        sh """
          #!/bin/bash
          whoami
          rm -rf /usr/share/nginx/html/build
        """
      }
    }

    stage('github clone (FE)') {
      steps {
        script {
          git branch: 'FE-Develop', credentialsId: 'gitlab_token',
          url: 'https://lab.ssafy.com/s11-bigdata-dist-sub1/S11P21A607.git'
        }
      }
    }

    stage('build (FE)') {
      steps {
        dir("./frontend/mutualrisk") {
          sh """
            node --version # 올바른 Node.js 버전이 출력되는지 확인
            yarn install
            CI=false yarn build
          """
        }
      }
    }

    stage('move build files to nginx static') {
      steps {
        sh """
          pwd
          ls -al
          mv frontend/mutualrisk/build /usr/share/nginx/html/
          cp -r /usr/share/nginx/html/stockImage /usr/share/nginx/html/build
          echo "Deploy React App success."
        """
      }
    }
  }

  post {
    always {
      script {
        def message
        if (currentBuild.result == 'SUCCESS') {
          message = "빌드 성공: ${env.JOB_NAME} #${env.BUILD_NUMBER} \n(<${env.BUILD_URL}|Details>)"
          mattermostSend (color: 'good',
            message: message,
            endpoint: 'https://meeting.ssafy.com/hooks/db9zfmfhbrjtgdbm3ozbi8e9o',
            channel: 'jenkins_build'
          )
        } else if (currentBuild.result == 'FAILURE') {
          message = "빌드 실패: ${env.JOB_NAME} #${env.BUILD_NUMBER} \n(<${env.BUILD_URL}|Details>)\n"
          mattermostSend (color: 'danger',
            message: message,
            endpoint: 'https://meeting.ssafy.com/hooks/db9zfmfhbrjtgdbm3ozbi8e9o',
            channel: 'jenkins_build'
          )
        } else {
          message = "빌드 상태: ${env.JOB_NAME} #${env.BUILD_NUMBER} \n(<${env.BUILD_URL}|Details>)"
          mattermostSend (
            message: message,
            endpoint: 'https://meeting.ssafy.com/hooks/db9zfmfhbrjtgdbm3ozbi8e9o',

```

```

        channel: 'jenkins_build'
    )
}
}
}
}
}
}
}

```

FastAPI 설정



FastAPI 구축 및 실행에 필요한 정보입니다.

디렉토리 구조

```

├── app
│   ├── __init__.py
│   ├── main.py
│   └── modules
│       ├── __init__.py
│       └── efficient_frontier_utils.py
├── Dockerfile
└── requirements.txt

```

Dockerfile

```

#
FROM python:3.9

#
WORKDIR /code

#
COPY ./requirements.txt /code/requirements.txt

#
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

#
COPY ./app /code/app

#
CMD ["uvicorn", "app.main:app", "--proxy-headers", "--host", "0.0.0.0", "--port", "80", "--log-level", "debug"]ubuntu@ip-172-26-3-190:~/fastapi$

```

효율적 프론티어 계산

▼ efficient_frontier_utils.py

```

import numpy as np
import pandas as pd

import pypfopt
from pypfopt.efficient_frontier import EfficientFrontier

def get_portfolio_info(portfolio):
    expected_returns = portfolio.expected_returns

    # expected_returns = [max(0, expected_return) for expected_return in expected_returns]

    cov_matrix = portfolio.cov_matrix
    # pandas DataFrame 형태로 변환
    cov_matrix = pd.DataFrame(cov_matrix)
    lower_bounds = portfolio.lower_bounds
    upper_bounds = portfolio.upper_bounds
    exact_proportion = portfolio.exact_proportion

    # 기대 수익률이 음수인 자산에 대한 제약 조건 완화
    for i, exp_return in enumerate(expected_returns):
        if exp_return < 0:
            # logger.info(f"자산 {i}의 기대 수익률이 음수입니다 ({exp_return}). 제
            # exact_proportion 제약 조건 해제

            # 임시 로직 : expected_return 절댓값 씌우기
            expected_returns[i] = abs(exp_return)

            exact_proportion[i] = None
            # 하한을 0으로 설정하여 투자하지 않을 수 있도록 함
            lower_bounds[i] = 0.0

    # 양의 준정수 확인
    eigenvalues = np.linalg.eigvals(cov_matrix)
    if not np.all(eigenvalues >= 0):
        raise ValueError("Covariance matrix is not positive semi-definite")

```

```

result = {}

# ef = EfficientFrontier(expected_returns, cov_matrix, weight_bounds=list(zip(lower_bounds, upper_bounds)))
# # exact_proportion을 반영하여 가중치 고정
# for i, proportion in enumerate(exact_proportion):
#     if proportion is not None: # exact_proportion이 None이 아닌 경우에만 처리
#         ef.add_constraint(lambda w, i=i, proportion=proportion: w[i] == proportion)

ef = EfficientFrontier(expected_returns, cov_matrix)
# lower_bounds 반영
for i in range(len(lower_bounds)):
    lower_bound = lower_bounds[i]
    upper_bound = upper_bounds[i]
    ef.add_constraint(lambda w: w[i] >= lower_bound)
    ef.add_constraint(lambda w: w[i] <= upper_bound)

for i, proportion in enumerate(exact_proportion):
    if proportion is not None: # exact_proportion이 None이 아닌 경우에만 처리
        ef.add_constraint(lambda w, i=i, proportion=proportion: w[i] == proportion)

ef_minvol = ef.deepcopy()
ef_maxret = ef.deepcopy()
ef_tmp = ef.deepcopy()

ef_tmp.max_sharpe()
# 샤프 비율 최대화 포트폴리오 가중치 가져오기
max_sharpe_weights = ef_tmp.clean_weights()

result['weights'] = [max_sharpe_weights[key] for key in max_sharpe_weights] # 결과에 가중치 저장

performance = ef_tmp.portfolio_performance(verbose=True)
expected_return, volatility, sharpe_ratio = performance

fictional_performance = {
    "expectedReturn": expected_return,
    "volatility": volatility,
    "sharpeRatio": sharpe_ratio
}

result['fictionalPerformance'] = fictional_performance
ef_minvol.min_volatility()
min_ret = ef_minvol.portfolio_performance()[0]
max_ret = ef_maxret._max_return()

print("expected_returns:", expected_returns)
print("min_ret:", min_ret)
print("max_ret:", max_ret)
print("exact proportion: ", exact_proportion)

ef_param_range = np.linspace(min_ret, max_ret - 0.0001, 100)
frontierPoints = []
for param_value in ef_param_range:
    ef_cur = ef.deepcopy()
    try:
        dict = ef_cur.efficient_return(param_value)
        expectedReturn, volatility, _ = ef_cur.portfolio_performance()
        weights = [dict[dic] for dic in dict]
        frontierPoint = {
            "expectedReturn": expectedReturn,
            "volatility": volatility,
            "weights": weights
        }
        frontierPoints.append(frontierPoint)
    except Exception as e:
        print(f"문제 발생 param_value: {param_value}")
        print(f"Error occurred: {e}")
        continue

result['frontierPoints'] = frontierPoints

return result

```

포트폴리오 추천 종목 선정

▼ main.py

```

from fastapi import FastAPI, Request
import numpy as np
import pandas as pd
from pypopt import EfficientFrontier
from pyspark.sql import SparkSession
from concurrent.futures import ThreadPoolExecutor, as_completed
import time

app = FastAPI()

# Spark 세션 생성
spark = SparkSession.builder \
    .appName("MySQL to Spark") \
    .config("spark.driver.extraClassPath", "/home/ubuntu/mysql-connector-j-9.0.0/mysql-connector-j-9.0.0.jar") \

```

```

.config("spark.sql.execution.arrow.pyspark.enabled", "true") \
.config("spark.driver.memory", "4g") \
.config("spark.executor.memory", "8g") \
.config("spark.executor.memoryOverhead", "1g") \
.config("spark.executor.cores", "2") \
.config("spark.driver.cores", "2") \
.config("spark.task.cpus", "1") \
.getOrCreate()

print("스파크 세션 생성")

# MySQL 연결 정보 설정
jdbc_url = "jdbc:mysql://j11a607.p.ssafy.io:3306/product"
properties = {
    "user": "ssafy",
    "password": "zeroticket607",
    "driver": "com.mysql.cj.jdbc.Driver"
}

def validate_and_convert_data(expected_returns, cov_matrix):
    try:
        # 기대 수익률과 공분산 행렬을 float64 타입으로 강제 변환
        expected_returns = np.array(expected_returns, dtype=np.float64)
        cov_matrix = np.array(cov_matrix, dtype=np.float64)
    except ValueError as e:
        print(f"Error in type conversion: {e}")
        raise ValueError("Data type conversion failed. Please ensure input is numerical.")

    return expected_returns, cov_matrix

# 샤프 비율 계산 함수
def get_efficient_frontier_performance(asset_id, expected_returns, cov_matrix, lower_bounds, upper_bounds, exact_proportion):
    # 데이터 검증 및 강제 변환
    expected_returns, cov_matrix = validate_and_convert_data(expected_returns, cov_matrix)

    # 양의 준정부 확인
    eigenvalues = np.linalg.eigvals(cov_matrix)
    if not np.all(eigenvalues >= 0):
        raise ValueError("Covariance matrix is not positive semi-definite")

    ef = EfficientFrontier(expected_returns, cov_matrix, weight_bounds=(lower_bounds, upper_bounds))
    # exact_proportion을 반영하여 가중치 고정
    for i, proportion in enumerate(exact_proportion):
        if proportion is not None: # exact_proportion이 None이 아닌 경우에만 처리
            ef.add_constraint(lambda w, i=i, proportion=proportion: w[i] == proportion)

    ef.max_sharpe() # 샤프 비율 최대화
    performance = ef.portfolio_performance(risk_free_rate=0, verbose=False)[:3]
    max_sharpe_weights = [weight for idx, weight in ef.clean_weights().items()]
    return {
        "newAssetId": asset_id,
        "expectedReturn": performance[0],
        "volatility": performance[1],
        "sharpeRatio": performance[2],
        "weights": max_sharpe_weights
    }

# 2. 공분산 행렬을 벡터화하여 생성하는 함수
def create_cov_matrix(portfolio, cov_matrix_dict):
    assets = np.array(portfolio)
    # 벡터화된 방식으로 공분산 행렬 생성
    cov_matrix = np.array([[cov_matrix_dict.get(asset_i, {}).get(asset_j, 0) for asset_j in assets]
                           for asset_i in assets])
    return pd.DataFrame(cov_matrix, index=portfolio, columns=portfolio)

# 기대 수익률 및 공분산 행렬 데이터 준비 함수
def prepare_data(portfolio, expected_returns_dict, cov_matrix_dict):
    expected_returns = np.array([expected_returns_dict[asset] for asset in portfolio])
    cov_matrix = create_cov_matrix(portfolio, cov_matrix_dict)
    return expected_returns, cov_matrix

# ThreadPool을 사용한 Sharpe Ratio 병렬 계산 함수
def compute_performance_weights(portfolio, expected_returns_dict, cov_matrix_dict, lower_bounds, upper_bounds, exact_proportion):
    expected_returns, cov_matrix = prepare_data(portfolio, expected_returns_dict, cov_matrix_dict)
    try:
        return get_efficient_frontier_performance(portfolio[-1], expected_returns, cov_matrix, lower_bounds, upper_bounds, exact_proportion)
    except Exception as e:
        print(f"Error calculating sharp ratio for portfolio {portfolio}: {e}")
        return None

# FastAPI 엔드포인트 - 클라이언트로부터 자산 리스트를 받음
@app.post("/optimize")
async def optimize_portfolio(request: Request):
    #테스트코드
    start_time = time.time()
    data = await request.json() # 클라이언트로부터 받은 데이터
    existing_assets = data["existing_assets"] # 기존 자산 리스트
    new_assets = data["new_assets"] # 새로운 자산 리스트
    lower_bounds = data["lower_bounds"] + [0]
    upper_bounds = data["upper_bounds"] + [1]
    exact_proportion = data["exact_proportion"] + [None]

    print("요청들어옴")

    # 필요한 자산을 목록 생성

```

```

relevant_assets = list(set(existing_assets + new_assets))
assets_str = ','.join(map(str, relevant_assets))

# 1. 기대 수익률 데이터 가져오기
start_ret_time=time.time()
sql_expected_returns = f"""
SELECT id, expected_return
FROM asset
WHERE id IN ({assets_str})
"""
expected_returns_df = spark.read.format("jdbc") \
    .option("url", jdbc_url) \
    .option("driver", "com.mysql.cj.jdbc.Driver") \
    .option("query", sql_expected_returns) \
    .option("user", properties["user"]) \
    .option("password", properties["password"]) \
    .load()

expected_returns_dict = {row['id']: row['expected_return'] for row in expected_returns_df.collect()}
print("기대 수익률 준비 완료")
end_ret_time=time.time()
print("기대수익률 데이터 가져오기 실행시간: ", end_ret_time-start_ret_time)

# 공분산 데이터 가져오기
start_covar_time = time.time()
id_list = [i for i in range(1, 93818596, 9687)]
id_list_str = ','.join(map(str, id_list))

sql_cov = f"""
SELECT asset_id_1, asset_id_2, covariance
FROM asset_covariance_real
WHERE asset_id_1 IN ({','.join(map(str, existing_assets))})
or id in ({id_list_str})
"""
cov_df = spark.read.format("jdbc") \
    .option("url", jdbc_url) \
    .option("driver", "com.mysql.cj.jdbc.Driver") \
    .option("query", sql_cov) \
    .option("user", properties["user"]) \
    .option("password", properties["password"]) \
    .load()

print("공분산 데이터 가져오기 완료")
end_covar_time = time.time()
print("공분산 데이터 가져오는 실행시간: ", end_covar_time-start_covar_time)

spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")
# covariance 필드를 float로 변환
cov_df = cov_df.withColumn("covariance", cov_df["covariance"].cast("float"))
to_pandas_start_time = time.time()
cov_df_pandas = cov_df.toPandas()
print("공분산 데이터를 pandas로 변환")
to_pandas_end_time = time.time()
print("공분산데이터를 pandas로 변환 실행시간: ",to_pandas_end_time-to_pandas_start_time)

# 공분산 행렬 생성
#테스트코드
start_time2 = time.time()
cov_matrix_dict = {}
for row in cov_df_pandas.itertuples(index=False):
    asset_1 = row.asset_id_1
    asset_2 = row.asset_id_2
    covariance_value = row.covariance

    if asset_1 not in cov_matrix_dict:
        cov_matrix_dict[asset_1] = {}
    cov_matrix_dict[asset_1][asset_2] = covariance_value

    if asset_2 not in cov_matrix_dict:
        cov_matrix_dict[asset_2] = {}
    cov_matrix_dict[asset_2][asset_1] = covariance_value
print("공분산 준비 완료")
#테스트코드
end_time2 = time.time()
print("공분산 행렬 생성실행시간: ",(end_time2-start_time2))

# 포트폴리오 조합 생성
start_time3 = time.time()
portfolios = [existing_assets + [new_asset] for new_asset in new_assets]
print("포트폴리오 조합 생성 완료")
end_time3 = time.time()
print("포트폴리오 조합 생성 실행 시간: ", end_time3-start_time3)

results = []

# ThreadPoolExecutor를 사용하여 병렬로 샵 비율 계산
#테스트코드
start_time1 = time.time()

with ThreadPoolExecutor(max_workers=8) as executor:
    futures = [executor.submit(compute_performance_weights, portfolio, expected_returns_dict, cov_matrix_dict, lower_bounds, upper_bounds, exact_pro

    print("포트폴리오 병렬 계산 완료")

    for future in as_completed(futures):

```

```

        result = future.result()
        if result:
            print(result)
            results.append(result)

#테스트코드
end_time1 = time.time()
print("사프비율 계산 실행시간: ", (end_time1-start_time1))

# 사프 비율이 높은 상위 5개 자산만 가져옴
top_5_assets = sorted(results, key=lambda x: -x['sharpeRatio'][:5])
print("결과반환", top_5_assets)

#테스트코드
end_time = time.time()
print("총 실행시간: ", (end_time-start_time))

return {"top_5_assets": top_5_assets}

```

환경변수 및 설정파일



개발 과정에서 사용한 설정파일 및 .env 파일 내용입니다

API Server application.yml 및 application-product.yml

application.yml

```

# nginx에서 제대로 매핑할 수 있게 하기 위해, swagger api 역시 /api/v1 하에서 동작해야 한다
springdoc:
  swagger-ui:
    path: /api/v1/swagger-ui

spring:
  application:
    name: mutualrisk

datasource:
  url: jdbc:mysql://j11a607.p.ssafy.io:3306/product?useUnicode=true&characterEncoding=UTF-8
  username: ssafy
  password: zeroticket607
  driver-class-name: com.mysql.cj.jdbc.Driver

jpa:
  hibernate:
    ddl-auto: none
  properties:
    hibernate:
      format_sql: true
      use_sql_comments: true
      dialect: org.hibernate.dialect.MySQLDialect
  jdbc:
    time_zone: Asia/Seoul

devtools:
  restart:
    additional-paths: .

data:
  redis:
    host: localhost
    port: 6379

  mongodb:
    uri: mongodb://ssafy:zeroticket607@j11a607.p.ssafy.io:27017/
    database: product

logging.level:
  org.hibernate.SQL: debug
  org.mongodb.driver: debug

jwt:
  secret-key: Z29nbY10bS1zZXJ2ZXItZGxyamVvYw9yb3JodG9kZ290c3Atam9vbmdhbmduaw0teWVvHNpbWpPaGFswvuqoxyve

oauth:
  kakao:
    client-id: ddd1cb07e0545f2d743ebe394aae68fa
    redirect-uri : https://j11a607.p.ssafy.io/login/kakao/callback
    auth: https://kauth.kakao.com
    api: https://kapi.kakao.com
    client-secret: Z1yVJxR6mF3RnLWh0kEDHazxeuQdiQe3
    scope: profile_nickname,profile_image, account_email

mail:
  host: smtp.gmail.com

```



```

port: 587
username: sujipark2009@gmail.com
password: waqo zlvt olao pwli
properties:
  mail:
    smtp:
      auth: true
      timeout: 5000
      starttls:
        enable: true

```

application-product.yml(배포용)

```

# nginx에서 제대로 매핑할 수 있게 하기 위해, swagger api 역시 /api/v1 하에서 동작해야 한다
springdoc:
  swagger-ui:
    path: /api/v1/swagger-ui

spring:
  application:
    name: mutualrisk

datasource:
  url: jdbc:mysql://j11a607.p.ssafy.io:3306/product?useUnicode=true&characterEncoding=UTF-8
  username: ssafy
  password: zeroticket607
  driver-class-name: com.mysql.cj.jdbc.Driver

jpa:
  hibernate:
    ddl-auto: none
  properties:
    hibernate:
      format_sql: true
      use_sql_comments: true

devtools:
  restart:
    additional-paths: .

data:
  redis:
    host: redis
    port: 6379

mongodb:
  uri: mongodb://ssafy:zeroticket607@j11a607.p.ssafy.io:27017/
  database: product

logging.level:
  org.hibernate.SQL: debug


jwt:
  secret-key: Z29nb310bS1zZXJ2ZXItZGxamVvYw9yb3JodG9kZ290c3Atam9vbmdbmdudaw0teWVvbHNpbWpkaGFswvuqoxyve

oauth:
  kakao:
    client-id: ddd1cb07e0545f2d743ebe394aae68fa
    redirect_uri : https://j11a607.p.ssafy.io/login/kakao/callback
    auth: https://kauth.kakao.com
    api: https://kapi.kakao.com
    client-secret: ZlyVJxR6mF3RnLWh0kEDHazxeuQdiQe3
    scope: profile_nickname,profile_image, account_email

mail:
  host: smtp.gmail.com
  port: 587
  username: sujipark2009@gmail.com
  password: waqo zlvt olao pwli
  properties:
    mail:
      smtp:
        auth: true
        timeout: 5000
        starttls:
          enable: true

```

Nginx 설정파일

 Nginx 프록시 설정 정보를 담은 파일입니다.

test.conf

```

server {
  listen 80; # 80포트로 받을 때
  server_name j11a607.p.ssafy.io; # 도메인주소

```

```

    return 301 https://j11a607.p.ssafy.io$request_uri;
}

server {
    listen 443 ssl http2;
    server_name j11a607.p.ssafy.io;

    # ssl 인증서 적용하기
    ssl_certificate /etc/letsencrypt/live/j11a607.p.ssafy.io/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/j11a607.p.ssafy.io/privkey.pem;

    location /api/v1 {
        proxy_pass http://localhost:8080/api/v1; # Request에 대해 어디로 리다이렉트하는지 작성. 8080 -> 자신의 springboot app 이사용하는 포트
        proxy_set_header Host $http_host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }


    location /fastapi/v1 {
        proxy_pass http://localhost:8081/fastapi/v1; # Request에 대해 어디로 리다이렉트하는지 작성. 8081 -> 자신의 fastapi 이용하는 포트
        proxy_set_header Host $http_host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    location /v3 { # swagger api에서, /v3/api-docs/swagger-config 경로로 api를 호출. 이 역시 spring으로 매핑시켜야 함
        proxy_pass http://localhost:8080/v3;
        proxy_set_header Host $http_host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    location / {
        root /usr/share/nginx/html/build;
        try_files $uri $uri/ /index.html;
        add_header 'Access-Control-Allow-Origin' '*';
        add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS';
        add_header 'Access-Control-Allow-Headers' 'DNT,User-Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Range';
        add_header 'Access-Control-Expose-Headers' 'Content-Length,Content-Range';
    }
}

```

Airflow 설정파일

 데이터 자동 적재를 위한 Airflow 코드입니다

1. Airflow docker image pull

```
docker pull apache/airflow
```

2. Dockerfile 생성

```

# 베이스 이미지로 apache/airflow:latest 사용
FROM apache/airflow:latest

# 시스템 패키지를 root 권한으로 설치 (필요시)
USER root
RUN apt-get update && apt-get install -y \
    build-essential \
    libssl-dev \
    libffi-dev \
    python3-dev \
    && apt-get clean

# airflow 사용자로 다시 변경
USER airflow

# 필요한 Python 패키지 설치
RUN pip install --no-cache-dir \
    connectorx \
    pandas \
    numpy \
    pymysql \
    tqdm \
    requests \
    finance-datareader \
    SQLAlchemy \
    yfinance \
    pytz \
    BeautifulSoup4 \
    prophet \
    plotly

```

3. Docker Image Build

```
docker build -t custom-airflow:latest
```

4. docker-compose.yaml

```
services:
  airflow-webserver:
    image: custom-airflow:latest
    environment:
      - AIRFLOW__CORE__LOAD_EXAMPLES=False
      - AIRFLOW__CORE__EXECUTOR=SequentialExecutor
      - AIRFLOW__DATABASE__SQL_ALCHEMY_CONN=postgresql+psycopg2://airflow:airflow@postgres:5432/airflow
      - AIRFLOW__CELERY__RESULT_BACKEND=db+postgresql://airflow:airflow@postgres:5432/airflow
      - AIRFLOW__SCHEDULER__SCHEDULER_HEARTBEAT_SEC=60
      - AIRFLOW__SCHEDULER__MIN_FILE_PROCESS_INTERVAL=60
      - AIRFLOW__SCHEDULER__MAX_THREADS=1
      - AIRFLOW__CORE__DEFAULT_TIMEZONE=Asia/Seoul
      - AIRFLOW__CORE__DEFAULT_UI_TIMEZONE=Asia/Seoul
    volumes:
      - ./dags:/opt/airflow/dags
      - ./logs:/opt/airflow/logs
      - ./plugins:/opt/airflow/plugins
    ports:
      - "8082:8080"
    user: "root"
    command: bash -c "airflow db init && airflow users create --username ssafy --firstname safy --lastname s --role Admin --email ssafy --password ssafy &&
    depends_on:
      - postgres

  airflow-scheduler:
    image: custom-airflow:latest
    environment:
      - AIRFLOW__CORE__LOAD_EXAMPLES=False
      - AIRFLOW__CORE__EXECUTOR=SequentialExecutor
      - AIRFLOW__DATABASE__SQL_ALCHEMY_CONN=postgresql+psycopg2://airflow:airflow@postgres:5432/airflow
      - AIRFLOW__CELERY__RESULT_BACKEND=db+postgresql://airflow:airflow@postgres:5432/airflow
      - AIRFLOW__SCHEDULER__SCHEDULER_HEARTBEAT_SEC=60
      - AIRFLOW__SCHEDULER__MIN_FILE_PROCESS_INTERVAL=60
      - AIRFLOW__SCHEDULER__MAX_THREADS=1
      - AIRFLOW__CORE__DEFAULT_TIMEZONE=Asia/Seoul
    volumes:
      - ./dags:/opt/airflow/dags
      - ./logs:/opt/airflow/logs
      - ./plugins:/opt/airflow/plugins
    user: "root"
    command: bash -c "airflow scheduler"
    restart: always
    depends_on:
      - postgres

  postgres:
    image: postgres:13
    environment:
      POSTGRES_USER: airflow
      POSTGRES_PASSWORD: airflow
      POSTGRES_DB: airflow
    volumes:
      - ./pgdata:/var/lib/postgresql/data
```

5. DAG 파일

예상수익률 데이터 적재

▼ ELT_expected_return.py

```
from datetime import datetime, timedelta
import pytz
import os
import sys
import urllib.request
import time
import json
import numpy as np
import pandas as pd
import re
from sqlalchemy import create_engine, text
import requests
from bs4 import BeautifulSoup
from tqdm import tqdm
from prophet import Prophet
import logging
import pymysql

from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator

# DB 접속 정보
user = 'ssafy'
password = 'zeroticket607'
host = 'j11a607.p.ssafy.io'
port = 3306
database = 'product'
```

```

# MatterMost 웹훅 URL (실패 시 알림 전송)
MATTERMOST_WEBHOOK_URL = "https://meeting.ssafy.com/hooks/wpd4y4hbeidu9nytpb6ha6jqzw"

# MatterMost 알림 함수 (실패 시 호출)
def notify_mattermost(context):
    requests.post(
        MATTERMOST_WEBHOOK_URL,
        json={
            "text": f"DAG {context['dag_run'].dag_id} 실패했습니다."
        }
    )

asset_name = None
expected_return_dic = {}

# DB 접속 엔진 객체 생성
def create_db_connection():
    try:
        engine = create_engine(f'mysql+pymysql://{user}:{password}@{host}:{port}/{database}?charset=utf8mb4')
        connection = engine.connect() # 연결 테스트
        print(">> MySQL DB 연결 성공!")
        return connection
    except Exception as e:
        print(f">> DB 연결 실패: {e}")
        exit(1)

# asset테이블을 데이터프레임으로 가져오기
def fetch_asset_data(**kwargs):
    try:
        connection = create_db_connection()
        # asset 테이블에서 데이터를 조회하여 asset_name DataFrame으로 저장
        query = "SELECT * FROM asset"
        asset_name = pd.read_sql(query, connection)
        print("asset_name: ", asset_name.head())
        print("asset_name의 len", len(asset_name))

        # 데이터프레임 확인
        if asset_name is not None and not asset_name.empty:
            print(">> Asset 테이블 조회 성공!")
            # XCom에 asset_name 전달
            kwargs['ti'].xcom_push(key='asset_name', value=asset_name)
        else:
            print(">> Asset 테이블에 데이터가 없습니다.")
            exit(1) # 데이터가 없을 경우 프로그램 종료

    except Exception as e:
        print(f">> DB 연결 실패: {e}")
        exit(1)
    finally:
        connection.close() # 연결 종료

# 종목별 예상 수익률 계산하기
def process_and_predict(**kwargs):
    # XCom에서 asset_name 받아오기
    asset_name = kwargs['ti'].xcom_pull(key='asset_name')
    expected_return_dic = {}

    # 1년(365일)의 초 계산 (1년 = 365 * 24 * 60 * 60 초)
    one_year_in_seconds = 365 * 24 * 60 * 60
    # 현재 시간의 Unix 타임스탬프
    current_timestamp = int(time.time())
    # 1년 전의 Unix 타임스탬프 계산
    one_year_ago_timestamp = current_timestamp - one_year_in_seconds
    # datetime으로 변환
    one_year_ago_datetime = datetime.fromtimestamp(one_year_ago_timestamp)

    # 각 종목마다 예상수익률 계산 후, expected_return_dic에 적재
    for i in tqdm(range(len(asset_name))):
        asset = asset_name.iloc[i]
        asset_id = asset.id

        # 쿼리 실행 및 결과를 DataFrame으로 저장
        try:
            connection = create_db_connection()
            # date 값이 1년 이내인 데이터만 가져오는 쿼리
            query = text("""
                SELECT * FROM asset_history
                WHERE asset_id = :asset_id
                AND date >= :one_year_ago_datetime
            """)
            # 쿼리 실행
            df = pd.read_sql(query, connection, params={
                "asset_id": asset.id,
                "one_year_ago_datetime": one_year_ago_datetime
            })
            print(">> 쿼리 실행 후, DataFrame에 저장 완료")

            # Unix timestamp를 YYYY-MM-DD 형식으로 변환
            df['ds'] = pd.to_datetime(df['date'], unit='s').dt.strftime('%Y-%m-%d')
            # daily_price_change_rate 컬럼을 y로 이름 변경
            df['y'] = df['daily_price_change_rate']
            # 필요한 컬럼만 남기기
            df_prophet = df[['ds', 'y']]

            # Prophet model 피팅하기
            print("Prophet fitting...")

```

```

        m = Prophet()
        m.fit(df_prophet)
        print("Prophet fit 완료")

        # Forecast
        future = m.make_future_dataframe(periods=252)
        print("Prophet make_future_dataframe 수행완료")
        forecast = m.predict(future)
        print("Prophet predict 수행완료")
        expected_returns = forecast['yhat'].mean()
        print("예상수익률 계산 완료")
        expected_return_dic[asset_id] = expected_returns

    except Exception as e:
        print(f">> 쿼리 실행 중 오류 발생: {e}")
    finally:
        # DB 연결 종료
        if 'connection' in locals() and connection is not None:
            connection.close()
            print(">> DB 연결 종료.")

# 예상 수익률 딕셔너리를 XCom에 저장
kwargs['ti'].xcom_push(key='expected_return_dic', value=expected_return_dic)

#expected_return_dic에 존재하는 종목별 예상수익률을 테이블에 적재하기
def update_expected_returns(**kwargs):
    # XCom에서 expected_return_dic 받아오기
    expected_return_dic = kwargs['ti'].xcom_pull(key='expected_return_dic')

    try:
        connection = create_db_connection()
        for asset_id, expected_return in tqdm(expected_return_dic.items()):
            query = text("""
                UPDATE asset
                SET expected_return = :expected_return
                WHERE id = :asset_id
            """)
            connection.execute(query, {'expected_return': expected_return, 'asset_id': asset_id})
        connection.commit()
        print(">> 모든 자산에 대해 expected_return 값을 업데이트했습니다.")

    except Exception as e:
        print(f">> DB 연결 실패: {e}")
        exit(1)

    finally:
        connection.close()
        print(">> DB 연결 종료.")

# DAG 정의
default_args = {
    'owner': 'ssafy',
    'start_date': datetime.now() - timedelta(hours=6),
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    'email_on_failure': False,
    'email_on_retry': False,
    'on_failure_callback': notify_mattermost, # 실패 시 MatterMost 알림
}

dag = DAG(
    "ELT_expected_return",
    default_args=default_args,
    description="종목별 예상 수익률 계산 ELT",
    schedule_interval="30 16 * * *", # 매일 1시 30분 실행
)

# ELT 작업의 각 태스크 정의
start = DummyOperator(
    task_id='start',
    dag=dag
)

fetch_asset_data_task = PythonOperator(
    task_id='fetch_asset_data_tasks',
    python_callable=fetch_asset_data,
    provide_context=True,
    dag=dag
)

process_and_predict_task = PythonOperator(
    task_id='process_and_predict_task',
    python_callable=process_and_predict,
    provide_context=True,
    dag=dag
)

update_expected_returns_task = PythonOperator(
    task_id='update_expected_returns_task',
    python_callable=update_expected_returns,
    provide_context=True,
    dag=dag
)

end = DummyOperator(
    task_id='end',
    dag=dag
)

```

```
)

# DAG 실행 순서 정의
start >> fetch_asset_data_task >> process_and_predict_task >> update_expected_returns_task >> end
```

주식 데이터 적재

▼ ETL_stock.py

```
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator
from airflow.providers.mysql.operators.mysql import MySQLOperator
from airflow.providers.http.operators.http import SimpleHttpOperator
from airflow.utils.dates import days_ago

import os
import time
import requests
from datetime import datetime, timedelta
import pandas as pd
import numpy as np
from tqdm import tqdm
from concurrent.futures import ThreadPoolExecutor, as_completed
from sqlalchemy import create_engine
from sqlalchemy.sql import text
import FinanceDataReader as fdr

# 데이터베이스 접속 정보 설정
DB_USER = 'ssafy'
DB_PASSWORD = 'zeroticket607'
DB_HOST = 'j11a607.p.ssafy.io'
DB_PORT = 3306
DB_DATABASE = 'raw_data'

# MatterMost 웹훅 URL (실제 시 알림 전송)
MATTERMOST_WEBHOOK_URL = "https://meeting.ssafy.com/hooks/wpd4y4hbeidu9nytpb6ha6jqzw"

# MatterMost 알림 함수 (실제 시 호출)
def notify_mattermost(context):
    requests.post(
        MATTERMOST_WEBHOOK_URL,
        json={
            "text": f"DAG {context['dag_run'].dag_id} 실패했습니다."
        }
    )

def fetch_stock_data(symbol):
    """주어진 심볼에 대해 데이터를 수집하고 처리"""
    try:
        # 심볼이 NaN이거나 잘못된 경우 스킵
        if not isinstance(symbol, str):
            return None

        # 오늘 날짜에서 하루를 뺀
        yesterday = (datetime.today() - timedelta(days=1)).strftime('%Y-%m-%d')

        # 데이터 수집
        df_temp = fdr.DataReader(symbol, yesterday)
        df_temp = df_temp[['Close']] # 추가 종가(Close)만 가져옴
        df_temp = df_temp.loc[[yesterday]]
        df_temp.index = pd.to_datetime(df_temp.index)
        df_temp.index.name = 'Date'
        df_temp.rename(columns={'Close': symbol}, inplace=True) # 컬럼명을 심볼로 변경
        return df_temp

    except Exception as e:
        print(f"Error fetching data for {symbol}: {e}")
        return None

def fetch_data_for_symbols(symbol_list, desc="Fetching stock data", max_workers=5):
    """심볼 리스트에 대해 데이터를 비동기로 수집"""
    # 결과를 저장할 리스트 초기화
    results_list = []

    # 스레드를 통해 심볼 리스트 병렬 처리
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = [executor.submit(fetch_stock_data, symbol) for symbol in symbol_list]

        # tqdm을 사용해 작업 진행도를 표시
        with tqdm(total=len(futures), desc=desc, unit="symbol") as pbar:
            for future in as_completed(futures):
                result = future.result()
                if result is not None:
                    results_list.append(result) # 병합 대신 리스트에 저장
                pbar.update(1) # 작업 완료 시 진행도 업데이트

    # 모든 결과를 리스트로 모아둔 후, 한 번에 병합
    if results_list:
        df_stocks = pd.concat(results_list, axis=1, join='outer') # 한 번에 병합
    else:
        df_stocks = pd.DataFrame() # 결과가 없을 경우 빈 데이터프레임 반환
```

```

return df_stocks.fillna(-1) # 결측치를 -1로 채움

def collect_krx_stock_data():
    """KOSPI, KOSDAQ, KONEX, ETF 데이터를 수집하고 병합"""
    # 종목 리스팅
    stocks_kospi = fdr.StockListing('KOSPI')
    kospi_code_list = stocks_kospi['Code'].tolist()

    stocks_kosdaq = fdr.StockListing('KOSDAQ')
    kosdaq_code_list = stocks_kosdaq['Code'].tolist()

    stocks_konex = fdr.StockListing('KONEX')
    konex_code_list = stocks_konex['Code'].tolist()

    # 데이터 수집
    df_kospi = fetch_data_for_symbols(kospi_code_list, desc="Fetching KOSPI data")
    df_kosdaq = fetch_data_for_symbols(kosdaq_code_list, desc="Fetching KOSDAQ data")
    df_konex = fetch_data_for_symbols(konex_code_list, desc="Fetching KONEX data")

    # ETF 데이터 수집
    etfs = fdr.StockListing('ETF/KR')
    etf_symbols = etfs['Symbol'].tolist()
    df_etfs = fetch_data_for_symbols(etf_symbols, desc="Fetching ETF data")

    # KRX 주식과 ETF 데이터를 합침
    krx_stock_with_etf = pd.concat([df_etfs, df_kospi, df_kosdaq, df_konex], axis=1)

    return krx_stock_with_etf, stocks_kospi, stocks_kosdaq, stocks_konex, etfs

def map_industry_categories(krx_list, etf_with_code_name):
    """KRX 종목에 산업 분류를 맵핑하고 ETF 데이터를 처리"""
    industry_category_map = {

        '측정, 시험, 항해, 제어 및 기타 정밀기기 제조업; 광학기기 제외': 'Industrials',
        '산업용 기계 및 장비 임대업': 'Industrials',
        '기타 금융업': 'Finance',
        '특수 목적용 기계 제조업': 'Industrials',
        '항공기, 우주선 및 부품 제조업': 'Industrials',
        '기타 전문 도매업': 'Consumer Discretionary',
        '컴퓨터 프로그래밍, 시스템 통합 및 관리업': 'Technology',
        '종합 소매업': 'Consumer Discretionary',
        '합성고무 및 플라스틱 물질 제조업': 'Basic Materials',
        '봉제외복 제조업': 'Consumer Discretionary',
        '자동차 신품 부품 제조업': 'Consumer Discretionary',
        '자연과학 및 공학 연구개발업': 'Technology',
        '영화, 비디오물, 방송프로그램 제작 및 배급업': 'Consumer Discretionary',
        '텔레비전 방송업': 'Telecommunications',
        '도로 화물 운송업': 'Industrials',
        '기타 식품 제조업': 'Consumer Staples',
        '음.식료품 및 담배 도매업': 'Consumer Staples',
        '의약품 제조업': 'Health Care',
        '건물 건설업': 'Industrials',
        '내화, 비내화 요업제품 제조업': 'Basic Materials',
        '통신 및 방송 장비 제조업': 'Telecommunications',
        '기타 화학제품 제조업': 'Basic Materials',
        '금융 지원 서비스업': 'Finance',
        '보험업': 'Finance',
        '반도체 제조업': 'Technology',
        '영상 및 음향기기 제조업': 'Consumer Discretionary',
        '전기업': 'Utilities',
        '상품 중개업': 'Consumer Discretionary',
        '기초 화학물질 제조업': 'Basic Materials',
        '토목 건설업': 'Industrials',
        '일차전지 및 이차전지 제조업': 'Industrials',
        '고무제품 제조업': 'Basic Materials',
        '신탁업 및 집합투자업': 'Finance',
        '1차 비철금속 제조업': 'Basic Materials',
        '1차 철강 제조업': 'Basic Materials',
        '기타 과학기술 서비스업': 'Technology',
        '전동기, 발전기 및 전기 변환 · 공급 · 제어 장치 제조업': 'Industrials',
        '부동산 임대 및 공급업': 'Real Estate',
        '생활용품 도매업': 'Consumer Staples',
        '소프트웨어 개발 및 공급업': 'Technology',
        '기타 섬유제품 제조업': 'Consumer Discretionary',
        '유원지 및 기타 오락관련 서비스업': 'Consumer Discretionary',
        '상품 종합 도매업': 'Consumer Discretionary',
        '전기 및 통신 공사업': 'Industrials',
        '일반 목적용 기계 제조업': 'Industrials',
        '건축기술, 엔지니어링 및 관련 기술 서비스업': 'Industrials',
        '선박 및 보트 건조업': 'Industrials',
        '의료용품 및 기타 의학 관련제품 제조업': 'Health Care',
        '전자부품 제조업': 'Technology',
        '해상 운송업': 'Industrials',
        '광고업': 'Consumer Discretionary',
        '기계장비 및 관련 물품 도매업': 'Industrials',
        '오디오물 출판 및 원판 녹음업': 'Consumer Discretionary',
        '유리 및 유리제품 제조업': 'Basic Materials',
        '시멘트, 석회, 플라스터 및 그 제품 제조업': 'Basic Materials',
        '자동차용 엔진 및 자동차 제조업': 'Consumer Discretionary',
        '그외 기타 운송장비 제조업': 'Industrials',
        '전기 통신업': 'Telecommunications',
        '담배 제조업': 'Consumer Staples',
        '기타 정보 서비스업': 'Technology',
        '플라스틱제품 제조업': 'Basic Materials',
        '무기 및 총포탄 제조업': 'Industrials',
        '사전장비 및 광학기기 제조업': 'Consumer Discretionary',
        '알코올음료 제조업': 'Consumer Staples',
    }

```

'자료처리, 호스팅, 포털 및 기타 인터넷 정보매개 서비스업': 'Technology',
 '서적, 잡지 및 기타 인쇄물 출판업': 'Consumer Discretionary',
 '회사 본부 및 경영 컨설팅 서비스업': 'Miscellaneous',
 '석유 정제품 제조업': 'Energy',
 '기타 비금속 광물제품 제조업': 'Basic Materials',
 '기타 운송관련 서비스업': 'Industrials',
 '자동차 부품 및 내장품 판매업': 'Consumer Discretionary',
 '기초 의약품 제조업': 'Health Care',
 '기록매체 복제업': 'Consumer Discretionary',
 '나무제품 제조업': 'Basic Materials',
 '교육지원 서비스업': 'Miscellaneous',
 '영상·오디오물 제공 서비스업': 'Consumer Discretionary',
 '철연선 및 케이블 제조업': 'Industrials',
 '기타 전기장비 제조업': 'Industrials',
 '스포츠 서비스업': 'Consumer Discretionary',
 '비료, 농약 및 살균, 살충제 제조업': 'Basic Materials',
 '가정용 기기 제조업': 'Consumer Discretionary',
 '연료용 가스 제조 및 배관공급업': 'Energy',
 '곡물·가공품, 전분 및 전분제품 제조업': 'Consumer Staples',
 '기타 사업지원 서비스업': 'Miscellaneous',
 '의복 액세서리 제조업': 'Consumer Discretionary',
 '기반조성 및 시설물 축조관련 전문공사업': 'Industrials',
 '실내건축 및 건축마무리 공사업': 'Industrials',
 '기타 종이 및 판지 제품 제조업': 'Basic Materials',
 '무점포 소매업': 'Consumer Discretionary',
 '그외 기타 전문, 과학 및 기술 서비스업': 'Miscellaneous',
 '전구 및 조명장치 제조업': 'Industrials',
 '은행 및 저축기관': 'Finance',
 '섬유, 의복, 신발 및 가죽제품 소매업': 'Consumer Discretionary',
 '펄프, 종이 및 판지 제조업': 'Basic Materials',
 '가구 제조업': 'Consumer Discretionary',
 '의료용 기기 제조업': 'Health Care',
 '동·식물성 유지 및 농축제품 제조업': 'Consumer Staples',
 '여행사 및 기타 여행보조 서비스업': 'Consumer Discretionary',
 '그외 기타 제품 제조업': 'Consumer Discretionary',
 '편조의복 제조업': 'Consumer Discretionary',
 '작물 재배업': 'Consumer Staples',
 '구조용 금속제품, 탱크 및 증기발생기 제조업': 'Industrials',
 '초등 교육기관': 'Miscellaneous',
 '금속 주조업': 'Basic Materials',
 '기타 금속 가공제품 제조업': 'Basic Materials',
 '연료 소매업': 'Energy',
 '굴판지, 종이 상자 및 종이용기 제조업': 'Basic Materials',
 '동물용 사료 및 조제식품 제조업': 'Consumer Staples',
 '직물직조 및 직물제품 제조업': 'Consumer Discretionary',
 '항공 여객 운송업': 'Industrials',
 '화학섬유 제조업': 'Basic Materials',
 '해체, 선별 및 원료 재생업': 'Utilities',
 '자동차 판매업': 'Consumer Discretionary',
 '육상 여객 운송업': 'Industrials',
 '도축, 육류 가공 및 저장 처리업': 'Consumer Staples',
 '어로 어업': 'Consumer Staples',
 '가죽, 가방 및 유사제품 제조업': 'Consumer Discretionary',
 '음식점업': 'Consumer Discretionary',
 '컴퓨터 및 주변장치 제조업': 'Technology',
 '일반 교습 학원': 'Miscellaneous',
 '운송장비 임대업': 'Industrials',
 '비알코올음료 및 알음 제조업': 'Consumer Staples',
 '가전제품 및 정보통신장비 소매업': 'Consumer Discretionary',
 '기타 교육기관': 'Miscellaneous',
 '일반 및 생활 숙박시설 운영업': 'Consumer Discretionary',
 '운동 및 경기용구 제조업': 'Consumer Discretionary',
 '인형, 장난감 및 오락용품 제조업': 'Consumer Discretionary',
 '수산물 가공 및 저장 처리업': 'Consumer Staples',
 '약기 제조업': 'Consumer Discretionary',
 '건설설비 설치 공사업': 'Industrials',
 '산업용 농·축산물 및 동·식물 도매업': 'Consumer Staples',
 '창작 및 예술관련 서비스업': 'Consumer Discretionary',
 '기타 생활용품 소매업': 'Consumer Staples',
 '전문디자인업': 'Miscellaneous',
 '기타 전문 서비스업': 'Miscellaneous',
 '섬유제품 염색, 정리 및 마무리 가공업': 'Consumer Discretionary',
 '인쇄 및 인쇄관련 산업': 'Miscellaneous',
 '경비, 경호 및 탐정업': 'Miscellaneous',
 '보험 및 연금관련 서비스업': 'Finance',
 '사업시설 유지·관리 서비스업': 'Miscellaneous',
 '시장조사 및 여론조사업': 'Miscellaneous',
 '자동차 재제조 부품 제조업': 'Consumer Discretionary',
 '마그네틱 및 광학 매체 제조업': 'Technology',
 '폐기물 처리업': 'Utilities',
 '과실, 채소 가공 및 저장 처리업': 'Consumer Staples',
 '제재 및 목재 가공업': 'Basic Materials',
 '귀금속 및 장신용품 제조업': 'Consumer Discretionary',
 '방적 및 가공사 제조업': 'Consumer Discretionary',
 '건축자재, 철물 및 난방장치 도매업': 'Consumer Discretionary',
 '편조원단 제조업': 'Consumer Discretionary',
 '증기, 냉·온수 및 공기조절 공급업': 'Utilities',
 '재 보험업': 'Finance',
 '그외 기타 개인 서비스업': 'Miscellaneous',
 '개인 및 가정용품 임대업': 'Consumer Discretionary',
 '부동산 관련 서비스업': 'Real Estate',
 '기타 상품 전문 소매업': 'Consumer Discretionary',
 '음·식료품 및 담배 소매업': 'Consumer Staples',
 '떡, 빵 및 과자류 제조업': 'Consumer Staples',
 '철도장비 제조업': 'Industrials',
 '신발 및 신발 부분품 제조업': 'Consumer Discretionary'

}


```

# KRX 전종목 목록 (설명 중심, 주식+펀드 등 전종목)
krx_desc_list = fdr.StockListing('KRX-DESC')

# Sector 컬럼과 맵 dict를 매핑하여 나온 결과를 새로운 컬럼에 추가
krx_desc_list['mappedSector'] = krx_desc_list['Sector'].map(industry_category_map)

# 섹터가 없는 데이터에 대해 -1로 분류
krx_desc_list = krx_desc_list.fillna(-1)

# 필요한 컬럼만 추출
krx_list_filtered = krx_desc_list[['Code', 'Name', 'mappedSector', 'Industry']]

# ETF들은 기본적으로 'ETF'라는 섹터로 분류
etf_with_code_name['mappedSector'] = 'ETF'
etf_with_code_name['Industry'] = 'ETF'

# ETF와 주식 데이터를 합침
krx_with_code_name = pd.concat([etf_with_code_name, krx_list_filtered], ignore_index=True)

# Region 추가
krx_with_code_name['Region'] = 'KR'

# 컬럼명 변경
krx_with_code_name = krx_with_code_name.rename(columns={'mappedSector': 'Sector'})

return krx_with_code_name

def merge_market_info(krx_with_code_name, stocks_kospi, stocks_kosdaq, stocks_konex, etfs):
    """국내 주식 정보에 시장 정보를 병합"""
    # 각 시장의 종목 정보를 합침
    df_kospi = stocks_kospi[['Code', 'Name', 'Market']]
    df_kosdaq = stocks_kosdaq[['Code', 'Name', 'Market']]
    df_konex = stocks_konex[['Code', 'Name', 'Market']]

    stock_with_code_name = pd.concat([df_kospi, df_kosdaq, df_konex], ignore_index=True)

    # ETF 정보 추가
    etf_with_code_name = etfs[['Symbol', 'Name']]
    etf_with_code_name['Market'] = 'ETF'
    etf_with_code_name = etf_with_code_name.rename(columns={'Symbol': 'Code'})

    # 주식과 ETF 정보를 합침
    stock_with_code_name = pd.concat([stock_with_code_name, etf_with_code_name], ignore_index=True)

    # Code를 기준으로 데이터프레임 병합
    merged_df = pd.merge(krx_with_code_name, stock_with_code_name[['Code', 'Market']], on='Code', how='left')

    # 컬럼명 정리
    krx_with_code_name = merged_df[['Code', 'Name', 'Sector', 'Industry', 'Region', 'Market']]

    return krx_with_code_name

def collect_nasdaq_data():
    """NASDAQ 주식 및 ETF 데이터를 수집하고 병합"""
    # NASDAQ 종목 데이터 읽기
    nasdaq_data = pd.read_csv('/opt/airflow/dags/nasdaq_screener2.csv', encoding='UTF-8', sep=',')
    nasdaq_data_filtered = nasdaq_data[['Symbol', 'Name', 'Sector']]
    nasdaq_data_filtered['Region'] = 'US'
    nasdaq_data_filtered['Market'] = 'NASDAQ'

    # NASDAQ ETF 데이터 읽기
    nasdaq_etf_data = pd.read_csv('/opt/airflow/dags/nasdaq_etf_screener.csv', encoding='UTF-8', sep=',')
    nasdaq_etf_data_filtered = nasdaq_etf_data[['SYMBOL', 'NAME']]
    nasdaq_etf_data_filtered['Sector'] = 'ETF'
    nasdaq_etf_data_filtered['Region'] = 'US'
    nasdaq_etf_data_filtered = nasdaq_etf_data_filtered.rename(columns={'SYMBOL': 'Code', 'NAME': 'Name'})
    nasdaq_etf_data_filtered = nasdaq_etf_data_filtered.drop(nasdaq_etf_data_filtered.index[-1]) # 이상한 데이터 제거
    nasdaq_etf_data_filtered['Market'] = 'ETF'
    nasdaq_etf_data_filtered['Industry'] = 'ETF'

    # 종목 리스트 생성
    nasdaq_symbol_list = nasdaq_data['Symbol'].tolist()
    nasdaq_symbol_list = [str(symbol) for symbol in nasdaq_symbol_list if isinstance(symbol, str)]
    nasdaq_symbol_list.extend(nasdaq_etf_data_filtered['Code'].tolist())

    # NASDAQ 주식 데이터 수집
    df_nasdaq = fetch_data_for_symbols(nasdaq_symbol_list, desc="Fetching NASDAQ data")

    # 결측치 처리
    df_nasdaq = df_nasdaq.fillna(-1)

    # CSV 파일과 실제 데이터의 차이를 보정하기 위해 필터링
    valid_symbols = set(nasdaq_symbol_list)
    columns_to_drop = [col for col in df_nasdaq.columns if col not in valid_symbols]
    df_nasdaq.drop(columns=columns_to_drop, inplace=True)

    # NASDAQ Industry 정보 가져오기
    df_nasdaq_with_industry = fdr.StockListing('NASDAQ')
    nasdaq_data_filtered = pd.merge(nasdaq_data_filtered, df_nasdaq_with_industry[['Symbol', 'Industry']], on='Symbol', how='left')
    nasdaq_data_filtered = nasdaq_data_filtered.rename(columns={'Symbol': 'Code'})
    nasdaq_data_filtered = nasdaq_data_filtered.fillna(-1)

    # NASDAQ ETF 산업정보 추가
    nasdaq_etf_data_filtered['Industry'] = 'ETF'

    # NASDAQ 주식과 ETF 데이터를 합침

```

```

nasdaq_total_with_code_name = pd.concat([nasdaq_data_filtered, nasdaq_etf_data_filtered], ignore_index=True)
nasdaq_total_with_code_name = nasdaq_total_with_code_name.fillna(-1)

return df_nasdaq, nasdaq_total_with_code_name

def merge_global_data(krx_with_code_name, nasdaq_total_with_code_name):
    """국내 주식 정보와 NASDAQ 정보를 합침"""
    total_with_code_name = pd.concat([krx_with_code_name, nasdaq_total_with_code_name], ignore_index=True)
    total_with_code_name = total_with_code_name.fillna(-1)
    return total_with_code_name

def remove_symbols_without_recent_data(df_data, df_prices):
    """최근 1년간 데이터가 없는 종목을 제거"""
    today = pd.to_datetime(datetime.now())
    one_year_ago = today - pd.DateOffset(years=1)

    recent_data = df_prices.loc[one_year_ago:]

    symbols_to_drop = recent_data.columns[(recent_data.iloc[0] == -1)]

    df_cleaned = df_prices.drop(columns=symbols_to_drop)

    # DataFrame에서 Code에 해당하는 값을 기준으로 필터링
    df_data_cleaned = df_data[~df_data['Code'].isin(symbols_to_drop)].reset_index(drop=True)

    return df_cleaned, df_data_cleaned

def fetch_krx_trend_data(codes):
    """
    주어진 KRX 종목 코드 리스트에 대해 트렌드 데이터를 수집합니다.
    """
    data_list = []

    # 오늘 날짜에서 하루를 뺀 날짜를 'YYYYMMDD' 형식으로 변환
    yesterday = datetime.today() - timedelta(days=1)
    bizdate = yesterday.strftime('%Y%m%d')

    for code in tqdm(codes, desc="Fetching trend data"):
        url = f'https://m.stock.naver.com/api/stock/{code}/trend?pageSize=5&bizdate={bizdate}'
        response = requests.get(url)

        if response.status_code == 200:
            data = response.json()

            for record in data:
                item = {
                    'Code': record.get('itemCode', ''),
                    'Bizdate': record.get('bizdate', ''),
                    'ForeignerPureBuyQuant': record.get('foreignerPureBuyQuant', ''),
                    'ForeignerHoldRatio': record.get('foreignerHoldRatio', ''),
                    'OrganPureBuyQuant': record.get('organPureBuyQuant', ''),
                    'IndividualPureBuyQuant': record.get('individualPureBuyQuant', ''),
                    'ClosePrice': record.get('closePrice', ''),
                    'AccumulatedTradingVolume': record.get('accumulatedTradingVolume', '')
                }
                data_list.append(item)
            else:
                print(f"Error fetching data for code {code}")

    df = pd.DataFrame(data_list)
    return df

def fetch_krx_stock_metrics(codes):
    """
    주어진 KRX 종목 코드 리스트에 대해 시가총액, PER, PBR, EPS 데이터를 수집합니다.
    """
    data_list = []

    for code in tqdm(codes, desc="Fetching stock metrics"):
        url = f'https://m.stock.naver.com/api/stock/{code}/integration'
        response = requests.get(url)

        if response.status_code == 200:
            data = response.json()

            # 시가총액, PER, PBR, EPS 초기화
            market_value = None
            per = None
            pbr = None
            eps = None

            total_infos = data.get("totalInfos", [])
            for info in total_infos:
                if info["code"] == "marketValue":
                    market_value = info.get("value", "")
                elif info["code"] == "per":
                    per = info.get("value", "")
                elif info["code"] == "pbr":
                    pbr = info.get("value", "")
                elif info["code"] == "eps":
                    eps = info.get("value", "")

            data_list.append({
                'Code': code,
                'MarketValue': market_value,
                'PER': per,
                'PBR': pbr,
            })

```

```

        'EPS': eps
    })
    else:
        print(f"Error fetching data for code {code}")

df = pd.DataFrame(data_list)
return df

def process_and_insert_krx_data(df_krx_cleaned, engine):
    """
    KRX 데이터를 처리하고 데이터베이스에 적재합니다.
    """
    df_melted = df_krx_cleaned.reset_index().melt(id_vars=["Date"], var_name="StockCode", value_name="Price")
    df_melted = df_melted[df_melted['Price'] != -1]
    df_melted.sort_values(by=['StockCode', 'Date'], inplace=True)
    df_melted['PriceChangePercent'] = df_melted.groupby('StockCode')['Price'].pct_change() * 100
    df_melted.replace([np.inf, -np.inf], 0, inplace=True)
    insert_data_to_db(df_melted, "krx_data", engine)

def process_and_insert_nasdaq_data(df_nasdaq_cleaned, engine):
    """
    NASDAQ 데이터를 처리하고 데이터베이스에 적재합니다.
    """
    df_nasdaq_cleaned.index.name = 'Date'
    df_melted = df_nasdaq_cleaned.reset_index().melt(id_vars=["Date"], var_name="StockCode", value_name="Price")
    df_melted = df_melted[df_melted['Price'] != -1]
    df_melted.sort_values(by=['StockCode', 'Date'], inplace=True)
    df_melted['PriceChangePercent'] = df_melted.groupby('StockCode')['Price'].pct_change() * 100
    df_melted.replace([np.inf, -np.inf], 0, inplace=True)
    insert_data_to_db(df_melted, "nasdaq_data", engine)

def insert_krx_stock_detail(df_stock_metrics, engine):
    """
    KRX 주식 세부사항 데이터를 데이터베이스에 적재합니다.
    """
    insert_data_to_db(df_stock_metrics, "krx_with_stock_detail", engine, drop_existing=True)

def insert_krx_trend(df_krx_trend, engine):
    """
    KRX 트렌드 데이터를 데이터베이스에 적재합니다.
    """
    insert_data_to_db(df_krx_trend, "krx_trend", engine, drop_existing=True)

def connect_to_db(user, password, host, port, database):
    try:
        engine = create_engine(f'mysql+pymysql://{user}:{password}@{host}:{port}/{database}?charset=utf8mb4')
        connection = engine.connect()
        print(">> MySQL DB 연결 성공!")
        return engine, connection
    except Exception as e:
        print(f">> DB 연결 실패: {e}")
        exit(1)

def insert_data_to_db(df, table_name, engine, drop_existing=False):
    connection = engine.connect()

    if drop_existing:
        try:
            connection.execute(text(f"DROP TABLE IF EXISTS {table_name}"))
            print(f">> 기존 테이블 '{table_name}' 삭제 완료")
        except Exception as e:
            print(f">> 테이블 삭제 실패: {e}")
            exit(1)

    total_rows = len(df)
    chunk_size = 10000

    try:
        with tqdm(total=total_rows, desc=f"'{table_name}' 테이블에 데이터 적재 중", unit="rows") as pbar:
            for i in range(0, total_rows, chunk_size):
                chunk = df.iloc[i:i + chunk_size]
                chunk.to_sql(index=False,
                              name=table_name,
                              con=engine,
                              if_exists='append',
                              method='multi',
                              chunksize=chunk_size)
                pbar.update(len(chunk))
            print(f">> '{table_name}' 테이블에 데이터 적재 성공!")
    except Exception as e:
        print(f">> 데이터 적재 실패: {e}")

def process():
    # 실행 시작 시간
    start_time = time.time()

    # 1. 국내 주식 및 ETF 데이터 수집

```

```

krx_stock_with_etf, stocks_kospi, stocks_kosdaq, stocks_konex, etfs = collect_krx_stock_data()

# 2. 산업 분류 맵핑 및 시장 정보 병합
krx_with_code_name = map_industry_categories(fdr.StockListing('KRX-DESC'), etfs[['Symbol', 'Name']].rename(columns={'Symbol': 'Code'}))
krx_with_code_name = merge_market_info(krx_with_code_name, stocks_kospi, stocks_kosdaq, stocks_konex, etfs)

# 3. NASDAQ 데이터 수집
df_nasdaq, nasdaq_total_with_code_name = collect_nasdaq_data()

# 4. 국내와 해외 데이터 합침
total_with_code_name = merge_global_data(krx_with_code_name, nasdaq_total_with_code_name)

# 5. 최근 1년간 데이터가 없는 종목 제거
df_nasdaq_cleaned, nasdaq_data_cleaned = remove_symbols_without_recent_data(nasdaq_total_with_code_name, df_nasdaq)
df_krx_cleaned, krx_data_cleaned = remove_symbols_without_recent_data(krx_with_code_name, krx_stock_with_etf)

# 6. 최종 데이터 합침
total_cleaned_with_code_name = pd.concat([krx_data_cleaned, nasdaq_data_cleaned], ignore_index=True)
total_cleaned_with_code_name = total_cleaned_with_code_name.reset_index(drop=True)

# 7. KRX 트렌드 및 주식 메트릭 데이터 수집
krx_codes = total_cleaned_with_code_name[
    (total_cleaned_with_code_name['Region'] == 'KR') & (total_cleaned_with_code_name['Industry'] != 'ETF')
]['Code'].tolist()

df_krx_trend = fetch_krx_trend_data(krx_codes)
df_stock_metrics = fetch_krx_stock_metrics(krx_codes)

# 8. 데이터베이스에 연결
engine, connection = connect_to_db(DB_USER, DB_PASSWORD, DB_HOST, DB_PORT, DB_DATABASE)

# 9. 데이터베이스에 데이터 적재
process_and_insert_krx_data(df_krx_cleaned, engine)
process_and_insert_nasdaq_data(df_nasdaq_cleaned, engine)
insert_krx_stock_detail(df_stock_metrics, engine)
insert_krx_trend(df_krx_trend, engine)

# 실행 종료 시간
cost_time_sec = time.time() - start_time
cost_time_min = cost_time_sec / 60
cost_time_hour = cost_time_min / 60

print(f">> 소요시간: {cost_time_sec:,.1f}초 = {cost_time_min:,.1f}분 = {cost_time_hour:,.1f}시간")

# 연결 종료
connection.close()

# DAG 정의
default_args = {
    'owner': 'ssafy',
    'start_date': days_ago(1),
    'email_on_failure': False,
    'email_on_retry': False,
    'on_failure_callback': notify_mattermost # 실패 시 MatterMost 알림
}

dag = DAG(
    "ETL_funds",
    default_args=default_args,
    description="주식 데이터 ETL",
    schedule_interval="0 15 * * *", # 매일 0시 15분 실행
    catchup=False
)

#Dummy Operator: ETL 시작
start = DummyOperator(task_id="start", dag=dag)

# DAG의 각 태스크 정의
krx_data_task = PythonOperator(
    task_id='process',
    python_callable=process,
    dag=dag
)

run_sql_queries = MySqlOperator(
    task_id='run_sql_queries',
    mysql_conn_id='ETL_stock', # Airflow connection ID를 사용
    sql="""
-- Asset History 테이블에 데이터 삽입
# 나스닥 종가기록 넣기
INSERT INTO asset_history (asset_id, date, price, daily_price_change_rate, `created_at`,`updated_at`)
SELECT
    a.id AS asset_id,
    n.date AS date,
    n.price AS price,
    IFNULL(round(n.priceChangePercent,2),0) as daily_price_change_rate,
    now(),
    now()
FROM
    asset a
JOIN
    raw_data.nasdaq_data AS n
ON a.code = n.StockCode
where n.Date = (SELECT MAX(Date) FROM raw_data.nasdaq_data);
    """
)

```

```

# krx 종가기록 넣기
INSERT INTO asset_history (asset_id, date, price, daily_price_change_rate, `created_at`,`updated_at`)
SELECT
    a.id AS asset_id,
    k.date AS date,
    k.price AS price,
    IFNULL(round(k.priceChangePercent,2),0) AS daily_price_change_rate,
    now(),
    now()
FROM
    asset a
JOIN
    raw_data.krx_data as k
ON a.code = k.StockCode
where k.Date = (SELECT MAX(Date) FROM raw_data.krx_data);

-- asset_history 테이블에서 daily_price_change_rate 업데이트
WITH price_changes AS (
    SELECT
        asset_id,
        id AS curr_id,
        price AS curr_price,
        LAG(price) OVER (PARTITION BY asset_id ORDER BY date) AS prev_price,
        ROW_NUMBER() OVER (PARTITION BY asset_id ORDER BY date DESC) AS rn
    FROM
        asset_history
)
UPDATE
    asset_history AS curr
JOIN
    price_changes AS pc ON curr.id = pc.curr_id
SET
    curr.daily_price_change_rate = ROUND(((pc.curr_price - pc.prev_price) / pc.prev_price) * 100, 2)
WHERE
    pc.rn = 1
    AND pc.prev_price IS NOT NULL;

-- asset 테이블의 recent_price와 oldest_price를 업데이트하는 쿼리
UPDATE asset a
JOIN (
    -- 가장 최근 가격을 가져오는 서브쿼리
    SELECT ah1.asset_id, ah1.price AS recent_price
    FROM asset_history ah1
    INNER JOIN (
        SELECT asset_id, MAX(date) AS max_date
        FROM asset_history
        WHERE status = TRUE
        GROUP BY asset_id
    ) ah2 ON ah1.asset_id = ah2.asset_id AND ah1.date = ah2.max_date
    WHERE ah1.status = TRUE
) recent ON a.id = recent.asset_id
JOIN (
    -- 가장 오래된 가격을 가져오는 서브쿼리
    SELECT ah3.asset_id, ah3.price AS oldest_price
    FROM asset_history ah3
    INNER JOIN (
        SELECT asset_id, MIN(date) AS min_date
        FROM asset_history
        WHERE status = TRUE
        GROUP BY asset_id
    ) ah4 ON ah3.asset_id = ah4.asset_id AND ah3.date = ah4.min_date
    WHERE ah3.status = TRUE
) oldest ON a.id = oldest.asset_id
SET
    a.recent_price = recent.recent_price,
    a.oldest_price = oldest.oldest_price;

-- raw_data.krx_trend 테이블 정제 및 데이터 처리
UPDATE raw_data.krx_trend
SET accumulatedTradingVolume = '0'
WHERE accumulatedTradingVolume = '-.';

UPDATE raw_data.krx_trend
SET accumulatedTradingVolume = REPLACE(accumulatedTradingVolume, ',', '')
WHERE accumulatedTradingVolume LIKE '%,%';

UPDATE raw_data.krx_trend
SET IndividualPureBuyQuant = REPLACE(IndividualPureBuyQuant, ',', '')
WHERE IndividualPureBuyQuant LIKE '%,%';

UPDATE raw_data.krx_trend
SET OrganPureBuyQuant = REPLACE(OrganPureBuyQuant, ',', '')
WHERE OrganPureBuyQuant LIKE '%,%';

UPDATE raw_data.krx_trend
SET ForeignerPureBuyQuant = REPLACE(ForeignerPureBuyQuant, ',', '')
WHERE ForeignerPureBuyQuant LIKE '%,%';

UPDATE raw_data.krx_trend
SET ForeignerHoldRatio = REPLACE(ForeignerHoldRatio, '%', '')
WHERE ForeignerHoldRatio LIKE '%%';

-- 국장 트렌드 데이터 삽입
DELETE FROM krx_stock_trend;

INSERT INTO krx_stock_trend (
    asset_id,

```

```

        code,
        date,
        foreigner_pure_buy_quant,
        foreigner_hold_ratio,
        organ_pure_buy_quant,
        individual_pure_buy_quant,
        accumulated_trading_volume,
        created_at,
        updated_at
    )
SELECT
    asset.id,
    krx_trend.code,
    STR_TO_DATE(krx_trend.bizdate, '%Y%m%d') AS date,
    CAST(REPLACE(krx_trend.foreignerPureBuyQuant, '+', '' ) AS SIGNED),
    CAST(REPLACE(krx_trend.foreignerHoldRatio, '+', '' ) AS DECIMAL(5, 2)),
    CAST(REPLACE(krx_trend.organPureBuyQuant, '+', '' ) AS SIGNED),
    CAST(REPLACE(krx_trend.individualPureBuyQuant, '+', '' ) AS SIGNED),
    CAST(REPLACE(krx_trend.accumulatedTradingVolume, '+', '' ) AS SIGNED),
    NOW(),
    NOW()
FROM
    raw_data.krx_trend AS krx_trend
JOIN
    asset ON asset.code = krx_trend.code;

-- 국장 상세 데이터 삽입
DELETE FROM krx_stock_detail;

INSERT INTO krx_stock_detail (
    asset_id,
    code,
    market_value,
    PER,
    PBR,
    EPS,
    created_at,
    updated_at,
    status
)
SELECT
    asset.id,
    stock_detail.code,
    stock_detail.marketValue,
    stock_detail.PER,
    stock_detail.PBR,
    stock_detail.EPS,
    NOW(),
    NOW(),
    TRUE
FROM
    raw_data.krx_with_stock_detail AS stock_detail
JOIN
    asset ON asset.code = stock_detail.code;

""",
dag=dag
)

#Dummy Operator: ETL 시작
end = DummyOperator(task_id="end", dag=dag)

# DAG 실행 순서 정의
start >> krx_data_task >> run_sql_queries >> end

```

[nasdaq_etf_screener.csv](#)

[nasdaq_screener2.csv](#)

구글뉴스 적재

▼ ETL_news_google.py

```

from datetime import datetime, timedelta
import pytz
import os
import sys
import urllib.request
import time
import json
import numpy as np
import pandas as pd
import re
from tqdm import tqdm
import requests
from bs4 import BeautifulSoup
from sqlalchemy import create_engine, text
from sqlalchemy.orm import sessionmaker
import math
import traceback

from airflow import DAG

```

```

from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator
from airflow.utils.dates import days_ago

# DB 접속 정보
user = 'ssafy'
password = 'zeroticket607'
host = 'j11a607.p.ssafy.io'
port = 3306
database = 'product'

# MatterMost 웹훅 URL (실패 시 알림 전송)
MATTERMOST_WEBHOOK_URL = "https://meeting.ssafy.com/hooks/wpd4y4hbeidu9nytpb6ha6jqzw"

# MatterMost 알림 함수 (실패 시 호출)
def notify_mattermost(context):
    requests.post(
        MATTERMOST_WEBHOOK_URL,
        json={
            "text": f"DAG {context['dag_run'].dag_id} 실패했습니다."
        }
    )

# 딕셔너리 형태의 뉴스 데이터를 적재할 리스트
news_data = []
# asset 테이블의 모든 name을 저장할 리스트
asset_name = []
# 뉴스의 url과 id를 매핑하는 딕셔너리
url_to_id_dic = {}

# 키워드로 검색 후, 뉴스 데이터 가져오는 코드
def google_news_scrape(keyword):
    global news_data

    # 구글 뉴스 검색 URL
    url = f'https://news.google.com/search?q={keyword}&when:1d&hl=en-NG&gl=NG&ceid=NG:en'

    # 웹 페이지 요청
    response = requests.get(url)
    if response.status_code != 200:
        print("Failed to retrieve the webpage")
        return

    # BeautifulSoup으로 페이지 파싱
    soup = BeautifulSoup(response.content, 'html.parser')

    # 뉴스 기사 정보 추출
    articles = soup.find_all('article')

    for article in articles:
        # 기사 제목 링크 및 URL 추출
        link_tag = article.find('a', href=True)
        if link_tag:
            article_url = 'https://news.google.com' + link_tag['href'][1:] # 상대 경로 수정

            # 제목 추출
            title_tag = article.find('a', class_='JtKRv')
            title = title_tag.get_text().strip()

            # 썸네일 이미지 추출
            figure_tag = article.find('figure', class_='K0q4G')
            if figure_tag:
                img_tag = figure_tag.find('img')
                if img_tag:
                    thumbnail_url = 'https://news.google.com' + img_tag.get('src', 'N/A')
                else:
                    thumbnail_url = 'No image found'
            else:
                thumbnail_url = 'No image found'

            # 날짜 추출
            time_tag = article.find('time')
            if time_tag:
                published_time = time_tag.get('datetime', 'N/A')
            else:
                published_time = 'N/A'

            news_data.append({
                'url': article_url,
                'published_time': published_time,
                'title': title,
                'thumbnail_url': thumbnail_url,
                'keyword': keyword
            })

def stock_name_task():
    global asset_name
    start_time = time.time()

    # DB 접속 엔진 객체 생성
    try:
        engine = create_engine(f'mysql+pymysql://{user}:{password}@{host}:{port}/{database}?charset=utf8mb4')
        connection = engine.connect()
        print(">> MySQL DB 연결 성공!")

        # asset 테이블에서 데이터를 조회하여 asset_name DataFrame으로 저장
        query = "SELECT name FROM asset"

```

```

        product_asset = pd.read_sql(query, connection)
        asset_name = product_asset['name'].tolist()
        print(">> asset 테이블에서 name 데이터 조회 성공!")

    except Exception as e:
        print(f">> DB 연결 실패: {e}")
        exit(1)

    finally:
        connection.close()

    cost_time_sec = time.time() - start_time
    cost_time_min = cost_time_sec / 60
    cost_time_hour = cost_time_min / 60
    print(f">> 소요시간: {cost_time_sec:.1f}초 = {cost_time_min:.1f}분 = {cost_time_hour:.1f}시간")

def news_data_task():
    global news_data
    global asset_name

    start_time = time.time()

    for name in tqdm(asset_name):
        google_news_scrape(name)

    cost_time_sec = time.time() - start_time
    cost_time_min = cost_time_sec / 60
    cost_time_hour = cost_time_min / 60
    print(f">> 소요시간: {cost_time_sec:.1f}초 = {cost_time_min:.1f}분 = {cost_time_hour:.1f}시간")

# news_test 테이블에 데이터 삽입
def news_insert_task():
    global news_data
    global url_to_id_dic
    start_time = time.time()

    # DB 접속 엔진 객체 생성 및 세션 생성
    engine = create_engine(f'mysql+pymysql://{user}:{password}@{host}:{port}/{database}?charset=utf8mb4')
    Session = sessionmaker(bind=engine)
    session = Session()

    # news_test 테이블에 데이터 삽입
    try:
        total_rows = len(news_data)
        with tqdm(total=total_rows, desc="뉴스 데이터 적재 중", unit="rows") as pbar:
            for news in news_data:
                thumbnail_url = news['thumbnail_url'] if news['thumbnail_url'] != 'No image found' else None
                published_time_str = news['published_time']
                try:
                    # ISO 8601 형식 날짜 변환
                    published_time = datetime.strptime(published_time_str, "%Y-%m-%dT%H:%M:%SZ")
                except ValueError as e:
                    print(f">> 경고: 날짜 형식을 파싱할 수 없습니다. ({published_time_str}) - {e}")
                    pbar.update(1)
                    continue

                # news_test 테이블에 데이터 삽입
                session.execute(text("""
                    INSERT INTO news_test (title, link, thumbnail_url, published_at, created_at, updated_at, status)
                    VALUES (:title, :link, :thumbnail_url, :published_at, now(), now(), 1)
                """), {
                    "title": news['title'],
                    "link": news['url'],
                    "thumbnail_url": thumbnail_url,
                    "published_at": published_time
                })
                last_id = session.execute(text("SELECT LAST_INSERT_ID()")).scalar()
                url_to_id_dic[news['url']] = last_id
                pbar.update(1)

            session.commit()
            print(">> news_test 데이터 적재 성공!")

    except Exception as e:
        print(f">> news_test 데이터 적재 실패: {e}")
        session.rollback()

# asset_news_test 테이블에 데이터 삽입
try:
    with tqdm(total=total_rows, desc="자산 데이터 적재 중", unit="rows") as pbar:
        for news in news_data:
            news_id = url_to_id_dic.get(news['url'])
            s = "INSERT INTO asset_news_test (news_test_id, asset_id, created_at, updated_at, status) VALUES (:news_id, :asset_id, now(), now(), 1)"
            asset_name = news.get('keyword')
            asset_id_query = session.execute(text("""
                SELECT id FROM asset WHERE name = :asset_name
            """), {"asset_name": asset_name}).scalar()

            if asset_id_query:
                asset_id = asset_id_query
                session.execute(text(s), {"news_id": news_id, "asset_id": asset_id})

        pbar.update(1)

    session.commit()
    print(">> asset_news_test 데이터 적재 성공!")

```



```

except Exception as e:
    print(f">> asset_news_test 데이터 적재 실패: {e}")
    print(traceback.format_exc())
    session.rollback()

finally:
    session.close()

cost_time_sec = time.time() - start_time
cost_time_min = cost_time_sec / 60
cost_time_hour = cost_time_min / 60
print(f">> 소요시간: {cost_time_sec:,.1f}초 = {cost_time_min:,.1f}분 = {cost_time_hour:,.1f}시간")

# DAG 정의
default_args = {
    'owner': 'ssafy',
    'start_date': days_ago(1),
    'retries': 0,
    'email_on_failure': False,
    'email_on_retry': False,
    'on_failure_callback': notify_mattermost, # 실패 시 MatterMost 알림
}

dag = DAG(
    "ETL_news_google",
    default_args=default_args,
    description="구글 뉴스 데이터 ETL",
    schedule_interval="30 15 * * *", # 매일 0시 30분 실행
)

# ETL 작업의 각 태스크 정의
start = DummyOperator(task_id="start", dag=dag)
stock_name_task = PythonOperator(task_id='stock_name_task', python_callable=stock_name_task, dag=dag)
news_data_task = PythonOperator(task_id='news_data_task', python_callable=news_data_task, dag=dag)
news_insert_task = PythonOperator(task_id='news_insert_task', python_callable=news_insert_task, dag=dag)
end = DummyOperator(task_id="end", dag=dag)

# DAG 실행 순서 정의
start >> stock_name_task >> news_data_task >> news_insert_task >> end

```

네이버 뉴스 적재

▼ ETL_news_naver.py

```

from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator
from airflow.utils.dates import days_ago

from datetime import datetime, timedelta
import pytz
import os
import sys
import urllib.request
import time
import json
import numpy as np
import pandas as pd
import re
from sqlalchemy import create_engine
import time
import requests
from bs4 import BeautifulSoup
from tqdm import tqdm
from sqlalchemy import text
import traceback
import ast
import math

# DB 접속 정보
user = 'ssafy'
password = 'zeroticket607'
host = 'j11a607.p.ssafy.io'
port = 3306
database = 'product'

# MatterMost 웹훅 URL (실패 시 알림 전송)
MATTERMOST_WEBHOOK_URL = "https://meeting.ssafy.com/hooks/wpd4y4hbeidu9nytpb6ha6jqzw"

# MatterMost 알림 함수 (실패 시 호출)
def notify_mattermost(context):
    requests.post(
        MATTERMOST_WEBHOOK_URL,
        json={
            "text": f"DAG {context['dag_run'].dag_id} 실패했습니다."
        }
    )

# 네이버 뉴스 api 관련 security 정보

```

```

client_id = "GKozhHirWV21sJdWZIA"
client_secret = "BIWjL3loMt"

def add_keyword_to_news(df):
    # key : url, value : 해당 뉴스의 url과 관련된 키워드
    url_keywords_dic = {}

    for idx, news in df.iterrows():
        # print(news_etf)
        url = news['link']
        if not url in url_keywords_dic:
            url_keywords_dic[url] = []
        if not pd.isna(news['search_keyword']):
            url_keywords_dic[url].append(news['search_keyword'])

    df_unique = df.drop_duplicates(subset='link', keep='first')
    df_unique['whole_keywords'] = [[] for _ in range(len(df_unique))]

    for idx, news in df_unique.iterrows():
        # print(news_etf)
        url = news['link']
        df_unique.at[idx, 'whole_keywords'] = url_keywords_dic[url]

    return df_unique

#[CODE 0]
# 네이버 api로 반환받은 데이터에서, &quot; 와 <b> 태그를 제거
def clean_text(input_text):
    # &quot; 제거
    cleaned_text = input_text.replace("&quot;", "")

    # <b>와 </b> 태그 제거
    cleaned_text = re.sub(r"<\/?b>", "", cleaned_text)

    return cleaned_text

#[CODE 1]
# 네이버 api를 요청하고, 응답을 반환하는 함수
def getRequestUrl(url):
    req = urllib.request.Request(url)
    req.add_header("X-Naver-Client-Id", client_id)
    req.add_header("X-Naver-Client-Secret", client_secret)

    try:
        response = urllib.request.urlopen(req)
        if response.getcode() == 200:
            # print("[%s]Url Request Success" % datetime.now())
            return response.read().decode('utf-8')
    except Exception as e :
        print(e)
        return None

#[CODE 2]
# 네이버 뉴스 api를 요청하고, 응답을 json 형태로 반환하는 함수
def getNaverSearch(node, srcText, start, display):
    base = "https://openapi.naver.com/v1/search"
    node = "/%.json" % node
    parameters = "?query=%s&start=%s&display=%s" % (urllib.parse.quote(srcText), start, display)

    url = base + node + parameters
    responseDecode = getRequestUrl(url)    #[CODE 1]

    if(responseDecode == None):
        return None
    else:
        return json.loads(responseDecode)

#[CODE 3]
# getNaverSearch가 반환한 결과를 저장
# naver news 형태로 되어 있고, 뉴스가 쓰여진 시각이 현재 시점 기준 4일 이내인 뉴스만 저장한다
def getPostData(post, jsonResult, cnt, srcText, asset_id, title1, pDate1, content1, link1, image_url1, keyword1):
    title = post['title']
    title = clean_text(title)
    org_link = post['originallink']
    link = post['link']

    pub_date = datetime.strptime(post['pubDate'], "%a, %d %b %Y %H:%M:%S %z")

    # 1일 전까지의 뉴스 데이터만 가져온다
    timezone = pytz.timezone('Asia/Seoul')
    today = datetime.now(timezone)
    one_days_ago = today - timedelta(days=1)

    if pub_date < one_days_ago:
        return False

    if "news.naver.com" not in link:
        return True

    pub_date = pub_date.strftime('%Y-%m-%d %H:%M:%S')

    (image_url, contents) = getNewsInfo(srcText, cnt, link)

    # 리스트에 값 추가
    title1.append(title)
    pDate1.append(pub_date)
    content1.append(contents)

```

```

link1.append(link)
image_url1.append(image_url)
keyword1.append(asset_id)

return True

#[CODE 4]
# 네이버 뉴스 api가 반환한 결과를 토대로, 직접 링크에 들어가 크롤링
# 뉴스 썸네일 이미지는
def getNewsInfo(keyword, cnt, link):
    response = requests.get(link,
                             headers={'User-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/128.0.0.0 Safari/!

    html = response.text
    soup = BeautifulSoup(html, "html.parser")

    # 기사 제목과 내용 추출
    contents = soup.select_one('#dic_area').get_text().strip() if soup.select_one('#dic_area') else '내용 없음'

    # 이미지 태그 추출
    img_tag = soup.find('img', id='img1')
    image_url = None

    if img_tag:
        image_url = img_tag['data-src']

        # # 이미지 다운로드
        # img_data = requests.get(image_url).content

        # image_name = f'{keyword}_{cnt}.jpg'

        # # 로컬에 이미지 저장 (경로 수정)
        # with open(f'C:\\Users\\SSAFY\\data-generate\\well_done data\\news_thumbnail\\{image_name}', 'wb') as handler:
        #     handler.write(img_data)

    return (image_url, contents)

#[CODE 0]
def main(asset_id, srcText, title1, pDate1, content1, link1, image_url1, keyword1):
    node = 'news' #크롤링한 대상
    # srcText = input('검색어를 입력하세요: ')
    cnt = 0
    jsonResult = []

    jsonResponse = getNaverSearch(node, srcText, 1, 100) #[CODE 2]
    total = jsonResponse['total']

    while((jsonResponse != None) and (jsonResponse['display'] != 0)):
        for post in jsonResponse['items']:
            cnt+= 1
            flag = getPostData(post, jsonResult, cnt, srcText, asset_id, title1, pDate1, content1, link1, image_url1, keyword1) #[CODE 3]
            if not flag:
                return

            start = jsonResponse['start'] + jsonResponse['display']
            # print("start : ", start)
            jsonResponse = getNaverSearch(node, srcText, start, 100) #[CODE 2]

def run_news_insert_dag():
    # DB 접속 엔진 객체 생성
    try:
        engine = create_engine(f'mysql+pymysql://{user}:{password}@{host}:{port}/{database}?charset=utf8mb4')
        connection = engine.connect() # 연결 테스트
        print(">> MySQL DB 연결 성공!")

        # asset 테이블에서 데이터를 조회하여 asset_name DataFrame으로 저장
        query = "SELECT * FROM asset"
        product_asset = pd.read_sql(query, connection)
        print(">> asset 테이블 조회 성공!")

    except Exception as e:
        print(f">> DB 연결 실패: {e}")
        exit(1)

    finally:
        connection.close() # 연결 종료

    title1 = []
    pDate1 = []
    content1 = []
    link1 = []
    image_url1 = []
    keyword1 = []

    # 1일 전까지의 뉴스 데이터만 가져온다

    timezone = pytz.timezone('Asia/Seoul')

    # iloc을 사용한 반복문 수정
    for i in tqdm(range(len(product_asset))): # DataFrame의 길이만큼 반복
        row = product_asset.iloc[i] # iloc을 사용해 각 row에 접근
        asset_id = row['id']
        name = row['name'] # 해당 row의 'name' 열 값 추출
        main(asset_id, name, title1, pDate1, content1, link1, image_url1, keyword1) # 추출한 name으로 main 함수 호출
        time.sleep(0.1) # API 요청 간격을 조절

    df=pd.DataFrame([title1, pDate1, content1, link1, image_url1, keyword1 ]).T

```

```

df.columns=['title','published_at', 'content', 'link', 'thumbnail_url', 'search_keyword']

df_unique = add_keyword_to_news(df)

url_to_id_dic = {}

# product data 삽입(news)

# 실행 시작 시간
start_time = time.time()

# DB 접속 엔진 객체 생성
try:
    engine = create_engine(f'mysql+pymysql://{user}:{password}@{host}:{port}/{database}?charset=utf8mb4')
    connection = engine.connect() # 연결 테스트
    print(">> MySQL DB 연결 성공!")
except Exception as e:
    print(f">> DB 연결 실패: {e}")
    exit(1)

# DB 테이블 명
table_name = "news_test"

# 데이터프레임
df = df_unique

# tqdm을 사용하여 진행 상황을 모니터링
total_rows = len(df)

# DB에 DataFrame 적재
try:
    with tqdm(total=total_rows, desc="데이터 적재 중", unit="rows") as pbar:
        for i in range(0, total_rows):
            # 데이터프레임을 chunksize 크기로 나누어 적재
            chunk = df.iloc[i]

            thumbnail_url = None if chunk.thumbnail_url is None or (isinstance(chunk.thumbnail_url, float) and math.isnan(chunk.thumbnail_url)) else
            connection.execute(text("INSERT INTO news_test (title, link, thumbnail_url, published_at, created_at, updated_at, status) VALUES (:title,
            {\"title\": chunk.title, \"link\": chunk.link, \"thumbnail_url\": chunk.thumbnail_url, \"published_at\": chunk.published_at})

            last_id = connection.execute(text("SELECT LAST_INSERT_ID()")).scalar()
            url_to_id_dic[chunk.link] = last_id
            # 진행 상황 업데이트
            pbar.update(1)
            print(">> 데이터 적재 성공!")
            connection.commit()
except Exception as e:
    print(f">> 데이터 적재 실패: {e}")
finally:
    connection.close()

# 실행 종료 시간
cost_time_sec = time.time() - start_time
cost_time_min = cost_time_sec / 60
cost_time_hour = cost_time_min / 60

print(f">> 소요시간: {cost_time_sec:.1f}초 = {cost_time_min:.1f}분 = {cost_time_hour:.1f}시간")

# product data 삽입(asset_news)

# 실행 시작 시간
start_time = time.time()

# DB 접속 엔진 객체 생성
try:
    engine = create_engine(f'mysql+pymysql://{user}:{password}@{host}:{port}/{database}?charset=utf8mb4')
    connection = engine.connect() # 연결 테스트
    print(">> MySQL DB 연결 성공!")
except Exception as e:
    print(f">> DB 연결 실패: {e}")
    exit(1)

# DB 테이블 명
table_name = "asset_test"

# 데이터프레임
df = df_unique

# tqdm을 사용하여 진행 상황을 모니터링
total_rows = len(df)

# DB에 DataFrame 적재
try:
    with tqdm(total=total_rows, desc="데이터 적재 중", unit="rows") as pbar:
        for i in range(0, total_rows):
            # 데이터프레임을 chunksize 크기로 나누어 적재
            chunk = df.iloc[i]
            news_id = url_to_id_dic[chunk.link]
            # SQL 쿼리와 바인딩할 값 리스트
            s = "INSERT INTO asset_news_test (news_test_id, asset_id, created_at, updated_at, status) VALUES (:news_id, :asset_id, now(), now(), 1)"
            values = []
            # 데이터프레임에서 값 추출
            # print(chunk.whole_keywords)
            for asset_id in chunk.whole_keywords:
                values.append({"news_id": news_id, "asset_id": asset_id})
            if values:
                connection.execute(text(s), values)

```

```

        pbar.update(1)
        print(">> 데이터 적재 성공!")
        connection.commit()
    except Exception as e:
        print(f">> 데이터 적재 실패: {e}")
        print(traceback.format_exc()) # 상세한 오류 메시지 출력
    finally:
        connection.close()

# 실행 종료 시간
cost_time_sec = time.time() - start_time
cost_time_min = cost_time_sec / 60
cost_time_hour = cost_time_min / 60

print(f">> 소요시간: {cost_time_sec:.1f}초 = {cost_time_min:.1f}분 = {cost_time_hour:.1f}시간")

# DAG 정의
default_args = {
    'owner': 'ssafy',
    'start_date': days_ago(1),
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    'email_on_failure': False,
    'email_on_retry': False,
    'on_failure_callback': notify_mattermost, # 실패 시 MatterMost 알림
}

dag = DAG(
    "ETL_news_naver",
    default_args=default_args,
    description="네이버 뉴스 데이터 ETL",
    schedule_interval="0 15 * * *", # 매일 0시 15분 실행
)

#Dummy Operator: ETL 시작
start = DummyOperator(task_id="start", dag=dag)

# DAG의 각 태스크 정의
news_insert_task = PythonOperator(
    task_id='news_insert_dag',
    python_callable=run_news_insert_dag,
    dag=dag
)

#Dummy Operator: ETL 시작
end = DummyOperator(task_id="end", dag=dag)

# DAG 실행 순서 정의
start >> news_insert_task >> end

```

외부 서비스

KAKAO OAuth 로그인에 사용

DB 덤프 파일 최신본

[DB 데이터 \(with Create Schema\).sql](#)

[DB 데이터 \(without Create Schema\).sql](#)