airflow-KPO-example

쿠버네티스파드오페레이터(kpo)예제를 설명과 함께 예제를 만들어 줘.(종류별로)

33s 동안 생각함

- K8s 1.31.1에서 Airflow 2.9.2 버전으로 KPO 예시를 제공해야 합니다. 다양한 종류를 다루며 간단한 BusyBox, 이미지 오버라이드, 리소스, 환경 변수, 비밀, 볼륨, 호스트 경로(NFS PVC) 등을 포함한 예시를 제공하고 서비스 계정, tolerations, nodeSelector 등도 다루겠습니다. 또한 Spark 작업 실행 예시도 포함하되, Spark 오퍼레이터 사용 여부에 대해서도 언급할 예정입니다.
- DAG 예시를 만들기 위해 기본적인 echo 작업부터 시작해서, 환경/비밀 설정, NFS PVC 마운트 및 uid 2001로 실행하도록 구성할 계획이에요. pod_template을 사용해 작업을 간소화하며, sidecar 컨테이너와 리소스, 포트를 설정할도 하고, XCom JSON 예시 및 동적 작업 매핑을 포함할 거예요. initContainers와 Helm 값 오버라이드를 예시로 보여줄게요.
- 비밀이 존재해야 한다는 점을 주의하며, NFS 경로에 PVC를 마운트하고 `runAsUser`와 `fsGroup`을 설정하여 작성할 수 있는 예시를 다룹니다. 또한 XCom을 통해 JSON 데이터를 기록하고, `pod_template_file`로 `initContainer`와 공유 볼륨 설정도 예시로 보여줄 예정입니다. 그런 다음 동적 작업 매핑과 Spark 제출 예시도 추가해 보겠습니다.

완료

좋아요. 운영 환경(airflow 2.9.2, K8s 1.31, RHEL9, ArgoCD/Helm, NFS)에 맞춰 “현업에서 바로 복붙→수정” 가능한 KPO 예제를 종류별로 정리했습니다. 각 스니펫은 **핵심 포인트 1~2개**만 보여주도록 작게 쪼갰어요. 필요하면 이걸 섞어 하나의 DAG로 합치면 됩니다.

---

0) 공통 준비 (imports / 기본 args)

```
# dags/kpo_examples_base.py
from datetime import datetime, timedelta
from airflow import DAG
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator
from kubernetes.client import models as k8s
```

```
default_args = {
    "owner": "data-platform",
    "retries": 1,
    "retry_delay": timedelta(minutes=2),
}

def make_dag(dag_id: str, schedule: str | None = None) -> DAG:
    return DAG(
        dag_id=dag_id,
        default_args=default_args,
        start_date=datetime(2024, 1, 1),
        schedule_interval=schedule,
        catchup=False,
        tags=["kpo", "examples"],
    )
```

---

1) 최소 예제 (echo 한 번 찍고 끝)

```
# dags/kpo_example_1_minimal.py
from kpo_examples_base import make_dag
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator

with make_dag("kpo_example_1_minimal") as dag:
    hello = KubernetesPodOperator(
        task_id="hello",
        name="hello",
        namespace="airflow",
        image="busybox:1.36",
        cmds=["/bin/sh", "-c"],
        arguments=["echo 'hello from KPO' && uname -a"],
        get_logs=True,
        is_delete_operator_pod=True,  # 작업 후 파드 삭제(클러스터 청결)
    )
```

포인트

- Helm/ArgoCD로 배포된 Airflow라면 `in_cluster=True`가 기본이라 kubeconfig 필요 없음.
- 빠르게 동작 확인할 때 쓰는 패턴.

2) 리소스/환경변수/이미지 고정

```
# dags/kpo_example_2_resources_env.py
from kpo_examples_base import make_dag
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator

with make_dag("kpo_example_2_resources_env") as dag:
    run_job = KubernetesPodOperator(
        task_id="run_job",
        name="run-job",
        namespace="airflow",
        image="python:3.11-slim",
        image_pull_policy="IfNotPresent",
        cmds=["python", "-c"],
        arguments=["import os; print('ENV', os.getenv('STAGE'))"],
        env_vars={"STAGE": "prod", "TZ": "Asia/Seoul"},
        container_resources={
            "request_memory": "256Mi",
            "request_cpu": "100m",
            "limit_memory": "1Gi",
            "limit_cpu": "500m",
        },
        get_logs=True,
        is_delete_operator_pod=True,
    )
```

포인트

- **`container_resources`**로 requests/limits 분리 지정.
- 운영 노드 과부하 방지와 스케줄링 안정화에 필수.

---

3) Secret/ConfigMap 주입 (envFrom + 파일 마운트)

> *사전 준비: `**kubectl -n airflow create secret generic my-secret --from-literal=DB_PWD=pass**`*
> *`**kubectl -n airflow create configmap my-conf --from-literal=APP_MODE=etl**`*

```python
# dags/kpo_example_3_secrets_config.py
from kpo_examples_base import make_dag
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator
from kubernetes.client import models as k8s

with make_dag("kpo_example_3_secrets_config") as dag:
    volume = k8s.V1Volume(
        name="secret-vol",
        secret=k8s.V1SecretVolumeSource(secret_name="my-secret")
    )
    mount = k8s.V1VolumeMount(
        name="secret-vol",
        mount_path="/opt/secret",  # 파일로 접근
        read_only=True
    )

    run = KubernetesPodOperator(
        task_id="use_secret_and_config",
        name="use-secret-and-config",
        namespace="airflow",
        image="busybox:1.36",
        cmds=["/bin/sh","-c"],
        arguments=[
            "echo $APP_MODE; echo $DB_PWD; ls -l /opt/secret; cat
/opt/secret/DB_PWD || true"
        ],
        env_from=[

k8s.V1EnvFromSource(config_map_ref=k8s.V1ConfigMapEnvSource(name="my-conf")),
            k8s.V1EnvFromSource(secret_ref=k8s.V1SecretEnvSource(name="my-
secret")),
        ],
        volumes=[volume],
        volume_mounts=[mount],
        get_logs=True,
        is_delete_operator_pod=True,
    )
```

포인트

- 같은 Secret을 **환경변수**와 **파일** 두 방식으로 동시에 접근하는 패턴.
- 자격증명은 가능하면 파일 마운트로(로그에 노출 위험 감소).

---

4) NFS/PVC 마운트 + 권한(UID 2001)로 쓰기

> *사전 준비: PVC `**nfs-datalake-pvc**` 가 있고, **NFS 서버 디렉토리의 소유자/권한이 UID=2001** 인 상황.*

```
# dags/kpo_example_4_pvc_uid2001.py
from kpo_examples_base import make_dag
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator
from kubernetes.client import models as k8s

with make_dag("kpo_example_4_pvc_uid2001") as dag:
    vol = k8s.V1Volume(
        name="datalake",

persistent_volume_claim=k8s.V1PersistentVolumeClaimVolumeSource(claim_name="nfs
-datalake-pvc")
    )
    mnt = k8s.V1VolumeMount(name="datalake", mount_path="/datalake")

    # 핵심: runAsUser/fsGroup을 NFS 쪽 UID/GID(=2001)로 맞춤
    sc = k8s.V1PodSecurityContext(run_as_user=2001, fs_group=2001)

    writer = KubernetesPodOperator(
        task_id="write_to_nfs_as_2001",
        name="write-to-nfs",
        namespace="airflow",
        image="busybox:1.36",
        cmds=["/bin/sh","-c"],
        arguments=[
            "id; touch /datalake/airflow_check.txt && echo 'ok' >
/datalake/airflow_check.txt && ls -l /datalake"
        ],
        security_context=sc,
        volumes=[vol],
        volume_mounts=[mnt],
        is_delete_operator_pod=True,
```

```
        get_logs=True,
    )
```

포인트

- NFS 소유/권한과 파드의 **runAsUser/fsGroup**을 맞춰야 “쓰기”가 안전합니다.
- DAG 내부에서만 권한 맞추고 싶다면 이렇게 KPO에만 적용하면 됩니다.

---

5) XCom으로 결과 받기 (정석 패턴)

```
# dags/kpo_example_5_xcom_return.py
import json
from kpo_examples_base import make_dag
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator

with make_dag("kpo_example_5_xcom_return") as dag:
    # KPO는 /airflow/xcom/return.json에 쓰면 XCom으로 수거합니다.
    producer = KubernetesPodOperator(
        task_id="produce_xcom",
        name="produce-xcom",
        namespace="airflow",
        image="python:3.11-slim",
        cmds=["python","-c"],
        arguments=[(
            "import json,os; "
            "os.makedirs('/airflow/xcom', exist_ok=True); "
            "json.dump({'status':'done','rows':123},
open('/airflow/xcom/return.json','w'))"
        )],
        do_xcom_push=True,            # 중요
        get_logs=True,
        is_delete_operator_pod=True,
    )

    # 예: 후속 태스크에서 XCom 사용 (PythonOperator 등)
    from airflow.operators.python import PythonOperator
    def _consume(ti):
        data = ti.xcom_pull(task_ids="produce_xcom")
        print("XCOM:", data)
```

```
    consumer = PythonOperator(
        task_id="consume_xcom",
        python_callable=_consume
    )

    producer >> consumer
```

포인트

- **경로 고정**: `/airflow/xcom/return.json`
- `do_xcom_push=True` 필수.

---

6) pod_template_file 사용 (initContainer, 세밀 제어)

> *템플릿 파일(YAML)을 사용하면 initContainer, hostAliases, annotation, seccomp 등 **파드 수준**의 디테일 제어가 깔끔해집니다.*

`include/pod-templates/kpo-init-template.yaml`

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: kpo-init
spec:
  restartPolicy: Never
  securityContext:
    runAsUser: 2001
    fsGroup: 2001
  initContainers:
    - name: init-fix-perm
      image: busybox:1.36
      command: ["/bin/sh","-c"]
      args: ["mkdir -p /datalake/job && chown -R 2001:2001 /datalake"]
      volumeMounts:
        - name: datalake
          mountPath: /datalake
  containers:
    - name: base
      image: python:3.11-slim
```

```
      command: ["python","-c"]
      args: ["print('run after init...')"]
      volumeMounts:
        - name: datalake
          mountPath: /datalake
  volumes:
    - name: datalake
      persistentVolumeClaim:
        claimName: nfs-datalake-pvc
```

DAG

```
# dags/kpo_example_6_pod_template.py
from kpo_examples_base import make_dag
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator

with make_dag("kpo_example_6_pod_template") as dag:
    job = KubernetesPodOperator(
        task_id="use_pod_template",
        name="use-pod-template",
        namespace="airflow",
        pod_template_file="/opt/airflow/include/pod-templates/kpo-init-
template.yaml",
        get_logs=True,
        is_delete_operator_pod=True,
    )
```

포인트

- 복잡한 파드 스펙은 YAML로 분리 → GitOps에 잘 맞음(ArgoCD로 함께 배포).
- DAG에는 “무엇을 실행할지”만 남겨 가독성↑

---

7) 노드 스케줄링 옵션 (tolerations/nodeSelector/affinity)

```
# dags/kpo_example_7_scheduling.py
from kpo_examples_base import make_dag
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator
```

```
from kubernetes.client import models as k8s

with make_dag("kpo_example_7_scheduling") as dag:
    tolerations = [k8s.V1Toleration(key="workload", operator="Equal",
value="etl", effect="NoSchedule")]
    node_selector = {"nodepool": "data"}
    affinity = k8s.V1Affinity(
        node_affinity=k8s.V1NodeAffinity(

required_during_scheduling_ignored_during_execution=k8s.V1NodeSelector(
                node_selector_terms=[
                    k8s.V1NodeSelectorTerm(
                        match_expressions=
[k8s.V1NodeSelectorRequirement(key="storage", operator="In", values=["nfs-
fast"])]
                    )
                ]
            )
        )
    )

    k = KubernetesPodOperator(
        task_id="pin_to_data_nodes",
        name="pin-to-data-nodes",
        namespace="airflow",
        image="busybox:1.36",
        cmds=["/bin/sh","-c"],
        arguments=["echo target data nodes; sleep 2;"],
        tolerations=tolerations,
        node_selector=node_selector,
        affinity=affinity,
        get_logs=True,
        is_delete_operator_pod=True,
    )
```

포인트

- **데이터 노드**로 워크로드 유도.
- 운영에서 “Spark/ETL 전용 풀”과 충돌 최소화.

---

8) 동적 매핑 (파티션 리스트를 병렬 KPO로 확장)

```
# dags/kpo_example_8_dynamic_mapping.py
from kpo_examples_base import make_dag
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator

PARTS = [f"2025-09-{d:02d}" for d in range(1, 6)]  # 예: 5개 파티션

with make_dag("kpo_example_8_dynamic_mapping") as dag:
    base = KubernetesPodOperator.partial(
        task_id="process_partition",
        namespace="airflow",
        image="busybox:1.36",
        cmds=["/bin/sh","-c"],
        get_logs=True,
        is_delete_operator_pod=True,
    ).expand(arguments=[["echo processing {{ ti.map_index }} {{ params.p }} &&
sleep 1"] for _ in PARTS],
             params=[{"p": p} for p in PARTS])
```

포인트

- **`.partial().expand()`**로 대량 파티션 병렬 처리.
- 각 맵 태스크에 파라미터/인덱스 템플릿 사용 가능.

---

9) Spark 작업을 KPO로 실행 (spark-submit 컨테이너)

> *Spark 클러스터/이미지 사양에 맞게 수정하세요.*

```
# dags/kpo_example_9_spark_submit.py
from kpo_examples_base import make_dag
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator

with make_dag("kpo_example_9_spark_submit", schedule=None) as dag:
    spark = KubernetesPodOperator(
        task_id="spark_submit_job",
        name="spark-submit-job",
        namespace="airflow",
        image="bitnami/spark:3.5.3",  # 내부 레지스트리 이미지로 교체 권장
```

```
        cmds=["/bin/bash","-lc"],
        arguments=[
            "spark-submit "
            "--master
k8s://https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT "
            "--deploy-mode cluster "
            "--name etl-job "
            "--conf spark.kubernetes.namespace=processing "
            "--conf spark.executor.instances=2 "
            "--conf spark.executor.cores=1 "
            "--conf spark.executor.memory=2g "
            "--conf spark.kubernetes.container.image=bitnami/spark:3.5.3 "
            "local:///opt/spark/examples/src/main/python/pi.py 1000"
        ],
        env_vars={"PYSPARK_PYTHON": "/usr/bin/python3"},
        get_logs=True,
        is_delete_operator_pod=True,
    )
```

포인트

- KPO로도 충분히 Spark 잡을 날릴 수 있음(전용 Operator 없이).
- 운영에선 **이미지/네임스페이스/서비스어카운트/Role** 정합 필수.

---

10) 서비스어카운트/이미지풀시크릿/어노테이션

```
# dags/kpo_example_10_sa_pullsecrets_anno.py
from kpo_examples_base import make_dag
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import
KubernetesPodOperator

with make_dag("kpo_example_10_sa_pullsecrets_anno") as dag:
    job = KubernetesPodOperator(
        task_id="sa_and_pullsecret",
        name="sa-and-pullsecret",
        namespace="airflow",
        service_account_name="airflow-kpo",      # 사전 생성 필요
        image="nexus.local:5000/etl/my-image:1.0",
        image_pull_secrets=["regcred"],          # 사전 생성 필요
        annotations={"cluster-autoscaler.kubernetes.io/safe-to-evict": "true"},
```

```
        cmds=["/bin/sh","-c"],
        arguments=["echo 'hi' && sleep 3"],
        get_logs=True,
        is_delete_operator_pod=True,
    )
```

포인트

- 사설 레지스트리/권한 분리 환경에서 거의 필수.

운영 팁 (짧고 정확하게)

- **로그 유실 방지**: `get_logs=True` + `is_delete_operator_pod=True`면 파드 종료 직후 삭제되므로, 에러 시 디버깅하려면 잠시 `is_delete_operator_pod=False`로.
- **권한 이슈(NFS)**: NFS export가 `root_squash`면 컨테이너 root라도 쓰기 제한. **runAsUser/fsGroup=2001**로 맞추거나 **initContainer에서 chown**.
- **이미지 차별화**: 공용 Airflow 이미지 그대로 두고, **KPO에서만 커스텀 이미지**(파이썬 모듈 포함) 쓰면 다른 DAG에 영향 없음.
- **리소스 한도**: requests/limits는 최소값부터. 한 번에 크게 잡으면 클러스터 스케줄 실패 → DAG 대기행 증가.
- **템플릿 분리**: 파드 스펙이 복잡해지면 `pod_template_file` 권장(ArgoCD로 함께 버전 관리).

필요하신 조합을 말씀해 주세요. 예를 들어 “NFS + Spark + 동적 매핑” 같은 **실전 DAG 한 파일**로 깔끔하게 묶어 드릴 수 있습니다.

"KPO로도 충분히 Spark 잡을 날릴 수 있음(전용 Operator 없이)." 그러면 왜 SsparkSubmitOperator 가 있는거야? 그리고 왜 사용하는거지?

짧게 핵심부터요.

- **가능은 함:** KPO로도 `spark-submit`를 컨테이너에서 실행하면 스파크 잡이 잘 돌아갑니다.

- **그런데도 SparkSubmitOperator가 존재하는 이유:** “스파크 잡만의 생명주기·옵션·취소/재시도 연동” 같은 **스파크 특화 편의성**을 Airflow 쪽에서 바로 제공하려고요.

아래를 보면 감이 옵니다.

언제 KPO로?

컨테이너/쿠버네티스 제어가 중요할 때요.

- 파드 스펙 세밀 제어: `securityContext(runAsUser=2001)`, `fsGroup`, `tolerations`, `affinity`, `initContainers`, PVC(NFS) 마운트.
- 사내 레지스트리/전용 이미지, 추가 파이썬 모듈, 네트워크/사이드카, 노드 핀닝 등 **K8s 레벨** 튜닝이 핵심일 때.
- Spark 외 작업(전처리/후처리, 파일 퍼미션 정리, 데이터 머지 등)과 **하나의 파드에서** 묶고 싶을 때.
- “내가 다 책임지고 제어할래” 모드. 로그 수집·상태 체크·클린업을 **직접 스크립트**로 넣어도 괜찮을 때.

> *단점*
>
> - *`spark-submit`가 띄운 **드라이버/익스큐터 파드 생명주기**가 Airflow Task와 느슨하게 연결되기 쉽습니다.*
>   - *KPO 파드가 끝나도 스파크 애플리케이션이 돌아갈 수 있음.*
>   - *작업 중지/재시도 시 **이전 스파크 앱 강제 종료**를 직접 구현해야 함(예: 드라이버 파드 delete).*
> - *잡 옵션(패키지, jars, conf) 관리·템플릿화는 **직접 스크립팅**해야 함.*

언제 SparkSubmitOperator로?

스파크 잡 자체의 생명주기/옵션 관리가 중요할 때요.

- Airflow에서 바로 **스파크 전용 인자**를 다루기 쉬움 (`application`, `application_args`, `conf`, `packages`, `jars`, `py_files`, `deploy_mode`, `num_executors`, `executor_memory` 등).
- **다중 런타임**을 같은 코드로 커버 (on-prem YARN/Standalone/K8s를 오가거나, 과거 워크로드가 섞여 있을 때 이점).
- **취소/재시도 시 정리**를 Airflow가 도와줌
  - 태스크 실패/킬 시 **기존 스파크 앱 종료** 로직이 내장(환경에 따라 동작 방식 차이는 있지만, 최소한 kill/cleanup 흐름을 내장).

- **로그/상태 모니터링 흐름**이 스파크 전용으로 단순화
  - submit 프로세스가 실패/성공으로 명확히 귀결, 리턴코드 처리 일관.

> *단점*
>
> - *파드 수준(UID 2001, PVC 마운트, initContainers 등)* ***쿠버네티스 디테일 제어가 약함.***
> - *스파크 말고* ***부수 작업****(NFS 권한 손보기, 외부 바이너리 준비 등)을 같은 태스크에서 함께 하긴 불편.*

현업 기준 빠른 의사결정 표

상황	추천
NFS에 UID=2001로 써야 함, initContainer로 퍼미션/데이터 세팅	**KPO**
팀이 스파크 옵션(jars/packages/conf)만 바꿔가며 돌림	**SparkSubmitOperator**
작업 중지/재시도 때 이전 스파크 앱을 깔끔히 끊어내야 함	**SparkSubmitOperator** (기본 흐름이 유리)
전/후처리(쉘·파이썬)와 스파크를 한 파드 안에서 붙이고 싶음	**KPO**
YARN/Standalone도 일부 남아 있어 이식성 필요	**SparkSubmitOperator**
쿠버네티스 세부 제어(노드 셀렉터, 네트워크, 사이드카)가 중요	**KPO**

하이브리드가 제일 실용적

- **전처리/권한/데이터 준비**: KPO (UID=2001, PVC 마운트, initContainers 등)
- **메인 스파크 잡**: SparkSubmitOperator (취소·재시도·옵션 관리 편의)
- **후처리/검증**: KPO 또는 PythonOperator

이렇게 쪼개면 각자 장점만 씁니다.

참고 스니펫(개념만)

1) KPO로 spark-submit + 안전 종료(드라이버 파드 삭제) 예시

```
# KPO 내 arguments 예시(개념)
"set -euo pipefail; \
 spark-submit ... --deploy-mode cluster ...; \
 STATUS=$?; \
 echo 'driver 정리 시그널(필요 시)'; \
 # kubectl delete pod <driver-pod> -n <ns> || true  # 상황 맞게 구현
 exit $STATUS"
```

- 취소/재시도 시 **이전 드라이버 파드 식별/삭제 로직**을 직접 넣어야 합니다.

2) SparkSubmitOperator로 스파크 옵션만 관리

```
from airflow.providers.apache.spark.operators.spark_submit import
SparkSubmitOperator

SparkSubmitOperator(
    task_id="spark_pi",
    application="local:///opt/spark/examples/src/main/python/pi.py",
    conf={
        "spark.kubernetes.namespace": "processing",
        "spark.executor.instances": "2",
        "spark.executor.cores": "1",
        "spark.executor.memory": "2g",
    },
    packages="org.postgresql:postgresql:42.7.3",
    name="etl-job",
    verbose=True,
)
```

- 스파크 팀이 익숙한 **spark-submit 인자 체계** 그대로 관리.

당신 환경에 딱 맞춘 권장

- **데이터레이크(NFS, UID=2001), Helm/ArgoCD, Airflow 2.9.2** 기준이면:
  1. **KPO로 전/후처리**(PVC 마운트, 퍼미션/파일 정리),
  2. **SparkSubmitOperator로 메인 잡**,
  3. 실패 시 **앱 클린업** 확인(드라이버 파드 잔류 여부 모니터링)
     이 3단 구성이 운영 안정성·가독성·유연성의 균형이 제일 좋습니다.

원하시면 “전처리(KPO) → 메인(SparkSubmitOperator) → 후처리(KPO)”를 **하나의 DAG 템플릿**로 바로 만들어 드릴게요.