

Airflow Spark Operator 주의점



SparkKubernetesOperator 를 사용하고 난뒤 가비지 configmap이 만개 정도가 있더라고.



34s 동안 생각함

- The user is facing leftover ConfigMaps after using the SparkKubernetesOperator, likely due to leftover Kubernetes submissions. I need to help with cleanup and prevention.
The SparkKubernetesOperator submitted via Airflow creates ConfigMaps for the application manifest or properties. Spark on Kubernetes also creates ConfigMaps for the driver/executor.
I'll provide Kubectl commands to list ConfigMaps, along with TTL suggestions to clean up these resources.
- In the Spark Operator, the 'sparkConfigMap' feature can mount an optional 'spark-conf-configmap', and resources like pods, services, and configmaps are labeled with 'spark-app-selector'. The operator can also create ConfigMaps to store driver configurations. To clean up stale ConfigMaps, you can use ``kubectl get cm -l spark-app-selector`` with label selectors like "sparkoperator.k8s.io/app-name" and "spark-role".
- To manage cleanup, I recommend setting up a script to delete old ConfigMaps not referenced by active pods. Use the ``timeToLiveSeconds`` in SparkApplication for automatic cleanup. You can also set ``deleteDriverPodOnFinish`` and ``deleteOnTermination`` for both driver and executor pods. Verify fields and labels to ensure proper cleanup.

🌐 웹에서 Spark Operator와 ConfigMap 정리 정보 검색

github.com

spark.apache.org

stackoverflow.com

13개 더 보기

- It seems that issue 1596 with the spark-operator involves missing ownerReferences, leading to ConfigMaps not being cleaned up. To address this, I could suggest upgrading to a version where the bug is fixed, but it looks like the issue might not have been addressed yet. I'll open the issue for further review.

🌐 문제 해결을 위한 spark-operator 업그레이드 확인

github.com



- The issue suggests that garbage collection isn't automatic for the created ConfigMaps in the Spark Operator. A potential solution is to set ownerReferences on ConfigMaps, though modifying the operator isn't simple. Alternatively, I can propose using label selectors like "spark-app-selector" to identify and delete stale ConfigMaps associated with completed or failed applications. I'll also suggest a script using `kubect1` and label selectors to automate this cleanup.
- I don't think enabling the 'metrics' webhook will fix the issue. It also seems that reducing ConfigMap creation with `spark.kubernetes.executor.podNamePrefix` isn't the solution. For cleanup, I should check if `spark.kubernetes.executor.deleteOnTermination` is set to true by default, as it might help with terminating executors cleanly.

🌐 Searching SparkExecutor config and termination options

[★ spark.apache.org](https://spark.apache.org)
[🔍 jaceklaskowski.github.io](https://github.com/jaceklaskowski)
[🔄 github.com](https://github.com)

👁️ 10개 더 보기

- So, I can suggest using a regular cleanup job to delete old ConfigMaps, labeling with `spark-app-selector` and `spark-role`. The command `kubect1 get cm -l 'spark-app-selector' -o json | jq` can filter ConfigMaps older than X days and delete them. Adding `ownerReferences` to ConfigMaps created by SparkApplications will guarantee proper deletion with cascading effects. Alternatively, reducing ConfigMap creation by configuring remote file storage could prevent the issue!
- 자원을 효율적으로 관리하려면, `spark.kubernetes.file.upload.path` 속성을 사용해 로컬 파일을 원격 저장소에 업로드하세요. 이렇게 하면 큰 ConfigMap을 생성할 필요가 없습니다. 또, SparkApplication의 `mode: cluster`와 `timeToLiveSeconds`

오... 그 “만 개” CM, 다들 한 번씩 당해요. 결론부터요: **Spark Operator/Native Spark가 생성한 임시 ConfigMap들이 자동 정리가 안 되는 케이스가 실제로 있습니다.** 특히 `spark-app-selector`, `spark-role` 라벨이 붙은 것들. 이건 이슈로도 보고됐고(“정리 안 됨, ownerReference 미부여”), 기본 동작만으론 남을 수 있어요. [GitHub](#)

아래처럼 **원인·예방·치우기**로 정리할게요.

왜 생기나

- Spark가 드라이버/익스큐터 구동 시 내부 설정·종속성을 **ConfigMap**으로 뿌리는 패턴을 씁니다. 이때 `spark-app-selector=...`, `spark-role=driver|executor` 같은 라벨이 달려요. 작업 끝나도 CM이 남는 사례가 있습니다. [GitHub](#)
- 파드/익스큐터는 `deleteOnTermination=true`로 잘 지워져도(ConfigMap은 별개) CM은 **가 비지 컬렉션 대상 연결(ownerReference)**이 없으면 남아있을 수 있어요. [Apache Spark](#) [Kubernetes](#)

앞으로 덜 생기게(예방)

1. 의존성 전달 방식을 CM 의존 ↓

- 드라이버/익스큐터 이미지에 JAR·파이썬 의존성을 **내장**하거나,
- ``spark.kubernetes.file.upload.path``(예: S3/HDFS/MinIO)로 로컬 파일 업로드 경로를 지정해 **ConfigMap** 대신 **원격 스토리지**를 쓰게 하세요. [Apache Spark](#)

2. SparkApplication TTL

- ``spec.timeToLiveSeconds``로 앱 종료 뒤 자원 정리 유도(완벽하진 않아도 Best Effort).
[Kubeflow](#)

3. 라벨 규칙 고정

- 우리 쪽에서 만드는 ConfigMap(예: ``spec.sparkConfigMap``)은 **ownerReferences**를 **SparkApplication**에 연결해 “연쇄 삭제”가 되게 설계. (K8s GC의 정석) [Kubernetes](#)
[Kubeflow](#)

지금 쌓인 것 치우기(안전 스크립트)

아래는 실행 중 파드가 참조 중인 CM은 건드리지 않고, Spark가 남긴 CM만 정리하는 순한맛 셸입니다.
(네임스페이스는 예시로 ``data-jobs``)

```
NS=data-jobs
DAYS=3 # N일 이전 것만 삭제

# 1) Spark가 만든 CM 후보 목록(라벨 기준)
kubectl -n $NS get cm -l 'spark-app-selector' -o json \
| jq -r '.items[]' \
| select(.metadata.creationTimestamp < (now - ("'$DAYS'"*86400) | todate)) \
| .metadata.name' \
| while read -r CM; do
# 2) 현재 어떤 파드가 이 CM을 마운트/참조 중인지 검사
IN_USE=$(kubectl -n $NS get pods -o json \
| jq -r --arg CM "$CM" '
[.items[]
| select(
( .spec.volumes[]? | select(.configMap and .configMap.name==$CM) )
or ( .metadata.annotations?["checksum/configmap-"+$CM] ) # 혹시나 헬
름 주입 패턴
) ] | length')

if [ "$IN_USE" -eq 0 ]; then
echo "deleting $CM"
kubectl -n $NS delete cm "$CM"
```

```

else
    echo "skip(in-use): $CM"
fi
done

```

라벨이 다르면 이 셀렉터를 바꾸면 됩니다. Spark가 남긴 CM은 흔히 이름 패턴이 ``spark-exec-* - conf-map`, `spark-* - driver-conf-map``이고, `**라벨 `spark-app-selector`, `spark-role`**`이 붙어 있어 골라내기 좋아요. [GitHub](#)

매일 자동화(CronJob)

위 스크립트를 컨테이너에 넣고 매일 새벽 돌리면 깔끔합니다.

```

apiVersion: batch/v1
kind: CronJob
metadata:
  name: spark-cm-janitor
  namespace: data-jobs
spec:
  schedule: "17 3 * * *" # 매일 03:17
  jobTemplate:
    spec:
      template:
        spec:
          serviceAccountName: janitor-sa
          containers:
            - name: janitor
              image: bitnami/kubect1:1.31
              command: ["/bin/bash", "-lc"]
              args:
                - |
                  set -euo pipefail
                  NS=data-jobs; DAYS=3
                  apk add --no-cache jq || true
                  kubectl -n $NS get cm -l 'spark-app-selector' -o json \
                  | jq -r '.items[]'
                  | select(.metadata.creationTimestamp < (now -
(env.DAYS|tonumber*86400) | todate))
                  | .metadata.name' \
                  | while read -r CM; do
                      IN_USE=$(kubectl -n $NS get pods -o json \
                      | jq -r --arg CM "$CM" '

```

```

        [.items[]
        | select(
            ( .spec.volumes[]? | select(.configMap and
            .configMap.name==$CM) )
        ) ] | length')
    if [ "$IN_USE" -eq 0 ]; then
        echo "deleting $CM"; kubectl -n $NS delete cm "$CM"
    else
        echo "skip(in-use): $CM"
    fi
done
restartPolicy: OnFailure

```

권한: `janitor-sa`에 `get/list/watch/delete` on `configmaps`, `get/list` on `pods` Role/RoleBinding 주시면 됩니다.

덧정리(체크포인트)

- 라벨 확인: `kubectl -n <ns> get cm -l spark-app-selector -o wide` 로 실제 분포부터 파악. [GitHub](#)
- **Executor/Driver** 파드 자동 삭제는 기본값이지만(ConfigMap과 별개), 혹시 튜닝 중 끄지 않았는지 `spark.kubernetes.executor.deleteOnTermination` 확인. [Apache Spark jaceklaskowski.github.io](#)
- 장기적으로는 “이미지 내장/원격경로 업로드(file.upload.path)”로 CM 의존도를 줄이는 게 제일 큼니다. [Apache Spark](#)

원하시면 당신 네임스페이스/라벨 규칙에 맞춰 더 공격적/더 보수적인 정리 스크립트 두 버전 만들어 드릴게요. 지금 상황(네임스페이스, 남아있는 CM 이름 패턴) 몇 줄만 알려주세요.



"권한: janitor-sa에 get/list/watch/delete on configmaps, get/list on pods Role/RoleBinding 주시면 됩니다." 이부분에 대해서 잘 모르겠어.



좋습니다 🍀. 이 부분은 “잡동사니 ConfigMap 치워주는 CronJob”이 실제로 kubectl 명령을 실행하려면 쿠버네티스 권한이 있어야 한다는 얘기였어요.

쿠버네티스에서는 파드가 뭘 할 수 있는지를 **ServiceAccount** → **Role/ClusterRole** → **RoleBinding/ClusterRoleBinding**으로 제어합니다.

단계별로 풀어보기

1. ServiceAccount (SA)

- 파드가 실행될 때 “내가 누구인지”를 나타내는 신분증 같은 것.
- ``default`` SA도 있지만, 보통은 목적별로 새 SA를 만들고 필요한 권한만 줍니다.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: janitor-sa
  namespace: data-jobs
```

2. Role

- “어떤 리소스(configmaps, pods)에 어떤 동작(get, list, delete 등)을 허용할까?” 정의.
- 네임스페이스 단위로 적용됩니다.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: janitor-role
  namespace: data-jobs
rules:
  - apiGroups: ["" ]
    resources: ["configmaps"]
    verbs: ["get", "list", "watch", "delete"]
  - apiGroups: ["" ]
    resources: ["pods"]
    verbs: ["get", "list"]
```

설명:

- ``apiGroups: [""]`` → Core API 그룹(= configmaps, pods 같은 기본 리소스).
- ``resources`` → 허용할 리소스 종류.
- ``verbs`` → 가능한 동작. (``get/list/watch/delete``)

3. RoleBinding

- “이 Role을 누구(ServiceAccount)에게 줄 것인가?” 연결.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: janitor-binding
  namespace: data-jobs
subjects:
  - kind: ServiceAccount
    name: janitor-sa
    namespace: data-jobs
roleRef:
  kind: Role
  name: janitor-role
  apiGroup: rbac.authorization.k8s.io
```

4. CronJob이 SA 사용

- 아까 만든 CronJob spec에서 `serviceAccountName: janitor-sa` 라고 적어주면, 그 잡은 janitor-sa의 권한을 갖게 됩니다.

큰 그림

- **CronJob 파드 → janitor-sa로 실행됨.**
- **janitor-sa는 janitor-role**을 RoleBinding으로 받아서 `configmaps`(get/list/watch/delete)와 `pods`(get/list) 권한을 가짐.
- 그래서 스크립트 안에서 `kubectl delete cm ...` 같은 명령이 에러 없이 동작.

혹시 원하시면, 제가 말씀드린 ① ServiceAccount, ② Role, ③ RoleBinding 세 개를 합쳐서 한 번에 적용할 수 있는 **all-in-one YAML** 만들어드릴까요?



항상 저렇게 delete 같은 작업을 할려면 이런식으로 신규로 만들어야 하는거야?



꼭 “항상 새로 만들어야 한다”는 건 아니에요 😊. 몇 가지 선택지가 있어요.

1. default ServiceAccount 그냥 쓰기

- 네임스페이스마다 `default`라는 SA가 이미 있어요.
 - 여기에 Role/RoleBinding을 연결해주면, CronJob에서 별도로 `serviceName`을 지정하지 않아도 권한을 가집니다.
 - **단점:** `default` SA를 공유하는 다른 파드들도 같은 권한을 얻게 돼요 → 원치 않는 삭제 권한이 퍼질 수 있음.
-

2. 기존 SA 재활용

- 운영 중 이미 권한을 가진 SA(예: `airflow`, `spark-operator` 등)가 있다면, 그 SA에 Role을 붙이거나 CronJob에 `serviceName: airflow`처럼 지정 가능.
 - **주의:** 해당 SA가 다른 용도로도 쓰이고 있다면, 불필요한 delete 권한이 섞여 들어갈 수 있음.
-

3. 전용 SA 만들기 (추천되는 관행)

- “정리용 CronJob”처럼 **특정한 목적**을 가진 워크로드에는 보통 전용 SA를 만들어요.
 - 이유: 최소 권한 원칙(Principle of Least Privilege). → 사고 났을 때 영향 범위를 좁힘.
 - 유지보수할 때도 “이 SA는 가비지 정리 전용”이라고 명확히 말할 수 있죠.
-

4. ClusterRole/ClusterRoleBinding

- 만약 여러 네임스페이스의 ConfigMap을 한꺼번에 청소해야 한다면 Role 대신 ClusterRole을 만들어야 합니다.
 - 이 경우는 권한 범위가 커지니 더 신중해야 해요.
-

✅ 정리하자면:

- **작은 실습/개인 환경:** default SA나 기존 SA에 권한 붙여도 무방.
- **운영 환경:** 전용 SA + Role/RoleBinding 조합이 제일 깔끔하고 안전합니다.

궁금한 점: 지금은 “내가 직접 테스트하는 Minikube/WSL2 환경”을 말씀하시는 건가요, 아니면 “운영 클러스터(SK하이닉스)”에서 돌릴 걸 염두에 두신 건가요?

환경에 따라 “default SA 써도 되는지 vs 전용 SA 필수인지” 판단이 달라집니다.



지금 운영되고 있는 클러스터에서 SA 부분을 파악할 필요가 있겠네.



맞습니다 👍

운영 클러스터에서 뭘 건드리기 전에 **지금 어떤 ServiceAccount들이 있고, 어디서 쓰이고 있는지를** 먼저 보는 게 제일 안전해요.

SA 현황 파악 기본 명령어

```
# 네임스페이스별 ServiceAccount 목록
kubectl get sa -n <네임스페이스>

# 특정 SA 상세 (어떤 Secret을 갖고 있는지, 어디 연결되어 있는지)
kubectl describe sa <SA이름> -n <네임스페이스>
```

SA ↔ Role/RoleBinding 관계 확인

```
# 네임스페이스 내 RoleBinding 확인
kubectl get rolebinding -n <네임스페이스>

# 특정 RoleBinding이 어떤 SA랑 연결되어 있는지 확인
kubectl describe rolebinding <RoleBinding이름> -n <네임스페이스>

# 어떤 권한을 갖고 있는지 Role/ClusterRole 조회
kubectl describe role <Role이름> -n <네임스페이스>
kubectl describe clusterrole <ClusterRole이름>
```

SA가 실제로 어디 파드에 쓰이는지 보기

```
# 파드 yaml 확인 → serviceAccountName 필드 체크
kubectl get pod <pod이름> -n <네임스페이스> -o yaml | grep serviceAccountName
```

체크 포인트

- **공유 SA** (예: `default`, `airflow`, `spark`)에 삭제 권한을 붙이면, 관련 없는 파드도 같은 권한을 가져버립니다.
- **전용 SA**를 만들어 주는 게 조금 번거롭더라도, “누구에게 어떤 권한을 줬는지” 추적하기가 훨씬 명확합니다.
- 운영에서는 특히 “삭제(delete)” 권한이 민감하니, 꼭 필요한 네임스페이스에만 좁게 줘야 합니다.

👉 먼저 운영 클러스터에서 `kubectl get sa -A` 찍어서 전체 그림을 보는 게 첫 단계예요. 혹시 원하시면, 제가 “운영 환경 SA 인벤토리 뽑는 스크립트”를 짧게 드릴까요? (네임스페이스별 SA, Role/RoleBinding 연결까지 한 번에 보는 형태)



응.



좋아요. 운영 클러스터에서 네임스페이스별 **ServiceAccount** 인벤토리 + 권한 연결 + 사용 중인 파드를 한 번에 뽑는 스크립트 드릴게요.

WSL2/리눅스 셸 기준이며 `kubectl`과 `jq`만 있으면 됩니다.

1) 초간단 훑어보기(사전 스캔)

```
# 모든 네임스페이스의 SA 수, RoleBinding 수, ClusterRoleBinding 수 대략 보기
kubectl get sa -A | wc -l
kubectl get rolebinding -A | wc -l
kubectl get clusterrolebinding | wc -l
```

2) 상세 인벤토리(CSV 출력)

- 무엇을 뽑나

```
`namespace, serviceaccount, bound_kind, bound_name, rbac_resource, rbac_verbs,
used_by_pods`
```

```
#!/usr/bin/env bash
set -euo pipefail

OUT="sa_inventory.csv"
echo
"namespace,serviceaccount,bound_kind,bound_name,rbac_resource,rbac_verbs,used_b
y_pods" > "$OUT"

# 네임스페이스 루프
for ns in $(kubectl get ns -o jsonpath='{.items[*].metadata.name}'); do
    # 이 NS의 SA 목록
    for sa in $(kubectl -n "$ns" get sa -o jsonpath='{.items[*].metadata.name}');
    do
        # 이 SA를 사용하는 파드 목록(공백 구분)
        pods=$(kubectl -n "$ns" get pods -o json | jq -r --arg SA "$sa" '
            [.items[] | select(.spec.serviceAccountName==$SA) | .metadata.name] |
            join(" ")')
        [[ -z "$pods" ]] && pods=""

        # 이 NS의 RoleBinding 중, 이 SA에 바인딩된 것 찾기 → 연결된 Role의 rules까지 펼치기
        kubectl -n "$ns" get rolebinding -o json | jq -r --arg SA "$sa" --arg NS
"$ns" '
            .items[]
            | select(.subjects[]? | select(.kind=="ServiceAccount" and .name==$SA and
(.namespace==$NS or (.namespace==null))))
            | {rb_name:.metadata.name, roleRefKind:.roleRef.kind,
roleRefName:.roleRef.name}
        ' | while read -r line; do
            [[ -z "$line" ]] && continue
            rb_name=$(echo "$line" | jq -r '.rb_name')
            role_kind=$(echo "$line" | jq -r '.roleRefKind')
            role_name=$(echo "$line" | jq -r '.roleRefName')

            if [[ "$role_kind" == "Role" ]]; then
                kubectl -n "$ns" get role "$role_name" -o json | jq -r --arg NS "$ns"
--arg SA "$sa" --arg pods "$pods" --arg rb "$rb_name" '
```

```

        .rules[] as $r |
        ($r.resources | join("&")) as $res |
        ($r.verbs | join("&")) as $verbs |
        [$NS, $SA, "RoleBinding", $rb, $res, $verbs, $pods] | @csv'
    elif [[ "$role_kind" == "ClusterRole" ]]; then
        kubectl get clusterrole "$role_name" -o json | jq -r --arg NS "$ns" -
-arg SA "$sa" --arg pods "$pods" --arg rb "$rb_name" '
        .rules[] as $r |
        ($r.resources | join("&")) as $res |
        ($r.verbs | join("&")) as $verbs |
        [$NS, $SA, "RoleBinding->ClusterRole", $rb, $res, $verbs, $pods] |
@csv'

    fi
done >> "$OUT"

# 이 SA가 ClusterRoleBinding을 통해 권한을 받는 경우(네임스페이스 무관)
kubectl get clusterrolebinding -o json | jq -r --arg SA "$sa" --arg NS
"$ns" '
    .items[]
    | select(.subjects[]? | select(.kind=="ServiceAccount" and .name==$SA and
.namespace==$NS))
    | {crb_name:.metadata.name, roleRefKind:.roleRef.kind,
roleRefName:.roleRef.name}
' | while read -r line; do
    [[ -z "$line" ]] && continue
    crb_name=$(echo "$line" | jq -r '.crb_name')
    role_kind=$(echo "$line" | jq -r '.roleRefKind')
    role_name=$(echo "$line" | jq -r '.roleRefName')

    # ClusterRoleBinding은 항상 ClusterRole을 가리킵니다
    kubectl get clusterrole "$role_name" -o json | jq -r --arg NS "$ns" --
arg SA "$sa" --arg pods "$pods" --arg crb "$crb_name" '
        .rules[] as $r |
        ($r.resources | join("&")) as $res |
        ($r.verbs | join("&")) as $verbs |
        [$NS, $SA, "ClusterRoleBinding", $crb, $res, $verbs, $pods] | @csv'
done >> "$OUT"

# Role/CRB 연결이 하나도 없었던 SA도 CSV에 남기고 싶다면 주석 해제:
# if ! grep -q "^\"$ns\", \"$sa\", \" \"$OUT\"; then
#     echo "\"$ns\", \"$sa\", \"(no-binding)\", \"\", \"\", \"\", \"$pods\" \" \" >>

```

```

"$OUT"
    # fi
done
done

echo "done: $OUT"

```

- 실행 후 현재 디렉터리에 `sa_inventory.csv`가 생깁니다.
- `rbac_resource`는 여러 리소스가 있을 수 있어 `&`로 묶었습니다. `verbs`도 동일.

3) “삭제 권한 보유 여부”만 빨리 체크(스팟 검사)

특정 네임스페이스에서, 각 SA가 **ConfigMap** 삭제 권한을 갖는지 바로 확인하고 싶으면:

```

NS=data-jobs
for SA in $(kubectl -n "$NS" get sa -o jsonpath='{.items[*].metadata.name}');
do
    if kubectl auth can-i delete configmaps --as=system:serviceaccount:$NS:$SA -n
"$NS" >/dev/null 2>&1; then
        echo "[CAN DELETE]  $NS/$SA"
    else
        echo "[NO DELETE ]  $NS/$SA"
    fi
done

```

주의: `kubectl auth can-i`는 규칙 기반 판단입니다. Admission Webhook 등 외부 정책이 있으면 실제 결과와 다를 수 있어요.

4) 눈으로 추적할 때 유용한 원-라이너들

```

# SA → 어떤 RoleBinding/ClusterRoleBinding에 묶였는지
NS=data-jobs; SA=airflow
kubectl -n $NS get rolebinding -o json \
| jq -r --arg SA "$SA" --arg NS "$NS" '.items[]
| select(.subjects[]? | select(.kind=="ServiceAccount" and .name==$SA and
(.namespace==$NS or (.namespace==null))))
| .metadata.name + " => " + .roleRef.kind + "/" + .roleRef.name'

kubectl get clusterrolebinding -o json \
| jq -r --arg SA "$SA" --arg NS "$NS" '.items[]
| select(.subjects[]? | select(.kind=="ServiceAccount" and .name==$SA and
.namespace==$NS))

```

```
| .metadata.name + " => " + .roleRef.kind + "/" + .roleRef.name'

# SA를 사용하는 파드
kubectl -n $NS get pods -o json | jq -r --arg SA "$SA" '.items[] |
select(.spec.serviceAccountName==$SA) | .metadata.name'
```

5) 결과 읽을 때 포인트

- **shared SA 여부**: 같은 SA를 여러 파드가 공유하면 권한도 같이 공유됩니다.
- **ClusterRoleBinding 유무**: 네임스페이스 경계를 넘는 권한이 들어와 있을 수 있습니다.
- **delete 권한 소유자**: 실제 “정리(CM 삭제)” 작업을 맡길 SA는 **전용 SA**로 두는 게 깔끔합니다.

필요하시면 위 CSV에서 “**delete configmaps** 권한을 가진 **SA만 필터링**” 해서 새 파일로 뽑는 후 처리 스크립트도 붙여드릴게요. 운영에서 바로 쓸 수 있게 네임스페이스 고정 버전으로도 만들어 드릴 수 있고요.