



우분투 리눅스에서 java jar 파일 하나 실행하는 pod 를 만들려고해. 같이 해보자.

좋아요, 바로 손 풀죠. "jar 하나 돌리는 Pod"는 용도가 둘 중 하나예요:

- 한 번 실행하고 끝나는 작업 → Job
- 계속 떠 있는 서버(예: Spring Boot) → Deployment(+ 선택: Service)

둘 다 템플릿 드릴게요. 사용하실 상황에 맞춰 골라 쓰시면 됩니다.

(WSL2 + Minikube 기준 명령도 같이 넣었습니다.)

0) 전제: 컨테이너 이미지에 JAR를 넣는 방식(권장)

Dockerfile

```
dockerfile

# 경량 JRE
FROM eclipse-temurin:17-jre
WORKDIR /app

# 로컬의 myapp.jar 를 이미지에 포함
COPY myapp.jar /app/app.jar

# 컨테이너 메모리 인식 & OOM 시 종료
ENTRYPOINT ["java", "-XX:MaxRAMPercentage=75%", "-XX:+ExitOnOutOfMemoryError", "-jar", "/app/app.jar"]
```

Minikube에서 이미지 빌드/적재

```
bash

# (1) Minikube 내부 Docker 데몬 사용
eval $(minikube -p minikube docker-env)

# (2) 이미지 빌드
docker build -t myjar:1.0 .

# (3) 네임스페이스 만들기(선택)
kubectl create ns apps

# (참고) 만약 외부 Docker 데몬에서 빌드했으면:
# minikube image load myjar:1.0 -p minikube
```

1A) “한 번 실행하고 끝” → Job 템플릿

```
yaml

apiVersion: batch/v1
kind: Job
metadata:
  name: myjar-job
  namespace: apps
spec:
  backoffLimit: 0                # 실패 시 재시도 안 함(원하면 조정)
  template:
    spec:
      restartPolicy: Never
      securityContext:
        runAsNonRoot: true
        runAsUser: 1000
      containers:
        - name: myjar
          image: myjar:1.0
          imagePullPolicy: IfNotPresent
          # java -jar 기본 진입점이라 args가 필요 없으면 생략 가능
          # args: ["--mode=batch", "--date=2025-09-01"]
          resources:
            requests:
              cpu: "100m"
              memory: "256Mi"
            limits:
              cpu: "1"
              memory: "1Gi"
```

실행/확인

```
bash

kubectl apply -f job.yaml
kubectl get jobs -n apps
kubectl logs job/myjar-job -n apps
```

1B) “계속 떠 있는 서버” → Deployment(+ Service) 템플릿

```
yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: myjar-deploy
  namespace: apps
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myjar
  template:
    metadata:
      labels:
        app: myjar
    spec:
      securityContext:
        runAsNonRoot: true
        runAsUser: 1000
      containers:
        - name: myjar
          image: myjar:1.0
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
```

```

        containerPort: 8080    # 서버가 리스닝하는 포트로 변경
env:
  - name: JAVA_TOOL_OPTIONS
    value: "-XX:MaxRAMPercentage=75 -XX:+ExitOnOutOfMemoryError"
resources:
  requests:
    cpu: "100m"
    memory: "256Mi"
  limits:
    cpu: "1"
    memory: "1Gi"
# 서버라면 헬스체크 권장 (HTTP 엔드포인트 맞춰 변경)
readinessProbe:
  httpGet:
    path: /actuator/health
    port: http
    initialDelaySeconds: 10
    periodSeconds: 5
livenessProbe:
  httpGet:
    path: /actuator/health
    port: http
    initialDelaySeconds: 30
    periodSeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: myjar-svc
  namespace: apps
spec:
  type: ClusterIP
  selector:
    app: myjar
  ports:
    - name: http
      port: 80
      targetPort: 8080

```

실행/접속(클러스터 내부)

```

bash

kubectl apply -f deploy-svc.yaml
kubectl get po,svc -n apps
kubectl logs -f deploy/myjar-deploy -n apps

```

로컬에서 테스트(Minikube)

```

bash

# 터널로 외부 접속
minikube service myjar-svc -n apps
# 또는 NodePort/Ingress로 노출하고 싶으면 타입만 바꾸거나 인그레스 추가

```

2) 이미지 안 만들고 JAR를 볼륨에서 실행하고 싶다면(대안)

NFS/iSCSI 등 PV에 jar가 이미 존재할 때 쓰는 패턴.

```

yaml

apiVersion: v1
kind: Pod

```

```

metadata:
  name: myjar-pod
  namespace: apps
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
  containers:
    - name: myjar
      image: eclipse-temurin:17-jre
      workingDir: /work
      command: ["java"]
      args: ["-XX:MaxRAMPercentage=75%", "-XX:+ExitOnOutOfMemoryError", "-jar", "/work/app.jar"]
      volumeMounts:
        - name: jar-vol
          mountPath: /work
      resources:
        requests:
          cpu: "100m"
          memory: "256Mi"
        limits:
          cpu: "1"
          memory: "1Gi"
  volumes:
    - name: jar-vol
      persistentVolumeClaim:
        claimName: myjar-pvc # 미리 만든 PVC

```

PVC만 적절히 연결하면 됩니다. 퍼미션 문제 생기면 PV 쪽 uid/gid 또는 **fsGroup** 조정해 주세요.

3) 자주 걸리는 포인트(현장 팁)

- **메모리 OOM**: 컨테이너 메모리 한도 내에서 JVM이 동작하도록 **-XX:MaxRAMPercentage** 사용. 애매하면 **requests**는 낮게, **limits**는 현실적으로.
- **헬스체크**: 서버라면 readiness/liveness 꼭. 엔드포인트 없으면 TCP 체크로 대체하거나 간단한 **/health** 추가.
- **이미지 배포**: 사내 레지스트리 쓰면 **imagePullSecrets** 설정. Minikube는 로컬 빌드가 빠르고 편합니다.
- **권한**: 기본 non-root 권장. 볼륨 퍼미션은 **fsGroup**으로 맞추면 편해요.
- **로그 보기**: `kubectl logs -f <pod>` 한 줄로 대부분 감 잡힙니다.

필요하시면 지금 바로 **Job**이든 **Deployment**든 실제 파일 이름으로 쪼개서(예: **job.yaml**, **deploy.yaml**, **svc.yaml**) 정리해 드릴게요.

수행자님 JAR 성격이 “배치 vs 서버” 중 어느 쪽인지 한마디만 알려주시면, 그에 맞춰 깔끔하게 마무리 세팅까지 붙여드립니다.



pv쪽 uid, gid에 대해서 무슨말인지 상세히 설명해줘.

좋아요. “PV쪽 uid/gid” 얘기는 결국 **파일 소유권** 문제예요.

컨테이너 안의 프로세스가 어떤 **UID(사용자 번호)/**GID(그룹 번호)****로 돌고, PV(파일시스템)에 있는 디렉터리/파일이 어떤 UID/GID로 **소유**되고 ****권한(mode)****이 어떻게 돼 있느냐, 이 조합이 맞아야 읽기/쓰기 됩니다.

1) 핵심 개념 한 장 요약

- **UID/GID = 숫자 기반 소유권**
리눅스는 “사용자 이름”이 아니라 **숫자(예: 1000:1000)**로 권한을 판단합니다.
- **컨테이너의 실행 UID/GID**
`securityContext.runAsUser`, `runAsGroup`, `fsGroup`, `supplementalGroups`로 결정됩니다. 이미지에 **USER 1000**이 박혀 있으면 그게 기본.
- **PV의 파일 소유권**
PV가 붙는 파일시스템(NFS, ext4/xfs, iSCSI 등) 쪽 디렉터리/파일에 이미 **소유자(UID/GID)와 퍼미션(예: 0750)**이 존재합니다.
- **불일치하면 Permission denied**
컨테이너 UID/GID가 PV의 권한 요건을 못 맞추면 쓰기 실패합니다.

2) Kubernetes에서 맞추는 방법 3가지

A. 컨테이너 쪽을 PV에 맞추기 (`runAsUser/runAsGroup`)

PV 디렉터리의 소유자가 1000:1000이라면, 컨테이너도 1000:1000으로 실행:

```
yaml

spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    runAsGroup: 1000
```

- **장점:** 빠르고 확실. 재귀 `chown` 안 해서 빠름.
- **주의:** 이미지 내부 파일/포트 바인딩 등 다른 부분과도 맞아야 합니다.

B. 그룹 권한으로 풀기 (`fsGroup`)

컨테이너의 주 그룹 또는 추가 그룹을 PV 디렉터리의 그룹으로 맞추고, 디렉터리가 그룹 쓰기를 허용하면 OK.

```
yaml

spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    runAsGroup: 1000
    fsGroup: 1000
    fsGroupChangePolicy: OnRootMismatch # 필요할 때만 재귀 권한 조정
```

- kubelet/CSI가 **볼륨 마운트 시** 디렉터리의 그룹 소유권/퍼미션을 조정하려 시도합니다.
- **대용량 볼륨**에서 `fsGroupChangePolicy: Always`는 느릴 수 있어요. `OnRootMismatch` 권장.
- 일부 드라이버/볼륨 타입은 적용 방식이 다릅니다(아래 4장 참고).

C. PV 쪽 소유권을 컨테이너에 맞게 바꾸기 (초기 1회 chown)

한 번만 초기화 컨테이너로 chown/chmod 해두고 이후 재사용:

```
yaml

initContainers:
- name: fix-perms
  image: busybox:1.36
  command: ["sh", "-c", "chown -R 1000:1000 /data && chmod -R g+rwX /data"]
  securityContext:
    runAsUser: 0          # chown 필요 시 root
  volumeMounts:
  - name: data
    mountPath: /data
```

- **장점:** 한 번만 맞춰두면 끝.
- **주의:** NFS root_squash 환경에선 root chown이 막힐 수 있습니다(아래 3장).

3) NFS(특히 root_squash)에서 자주 터지는 케이스

- **root_squash**가 기본이면, 컨테이너에서 root(0)로 접근해도 NFS 서버가 **익명 유저(보통 65534:65534 = nobody:nogroup)**로 바꿔버립니다. 그 상태에서 chown/chmod가 안 먹어서 Permission denied.
- 해결 옵션:
 1. 서버에서 디렉터리 소유권을 미리 맞춤: NFS 서버에서

```
bash

chown -R 1000:1000 /export/myshare
chmod -R 0770 /export/myshare
```

클라이언트(컨테이너)는 runAsUser: 1000, fsGroup: 1000으로 접근.

2. NFS export에 anonuid/anongid 지정:

```
/srv/share 10.0.0.0/24(rw,root_squash,anonuid=1000,anongid=1000)
```

이렇게 익명 사용자도 1000:1000으로 매핑.

3. root_squash 해제(no_root_squash): 보안상 비권장. 내부 폐쇄망이라도 신중.

- **요점:** NFS는 “숫자 UID/GID 일치”가 생명입니다. 이름이 아니라 숫자를 맞추세요.

4) 드라이버/볼륨 타입별 메모

• CSI 드라이버의 fsGroupPolicy

kubectl get csidriver -o yaml로 보면 spec.fsGroupPolicy가 있습니다.

드라이버가 kubelet의 재귀 chown을 대신 처리하기도 하고(File), 아예 안 하기도(None).

→ 드라이버 문서에서 권장값/동작 확인 필요.

• block → ext4/xfs 포맷되는 PV(iSCSI/Longhorn 등)

파일시스템에 박힌 소유권이 지속됩니다. 첫 마운트에만 chown 한 번 하고 끝내는 패턴이 좋습니다.

- **hostPath**
호스트의 디렉터리 권한 그대로 따라갑니다. 노드마다 다르면 골칫거리. 가능하면 피하고, 쓰면 노드 측 권한 정렬.
- **ConfigMap/Secret 볼륨**
소유자 **root**, 모드는 **defaultMode**(예: 0640). **fsGroup**을 주면 **group 읽기**가 보통 가능해집니다.
- **subPath 사용 시**
예전엔 **fsGroup**이 서브패스에 재귀 적용되지 않는 이슈가 있었습니다. 지금도 드라이버/커널 조합에 따라 **권한 자동 조정 안 먹는** 경우가 있어요. **subPath** 쓰면 **PV 측 권한을 사전에 맞추는** 게 안전합니다.

5) 현장용 체크리스트

1. 컨테이너에서 확인

```
bash

id -u; id -g; id
stat -c '%u:%g %a %n' /mnt/data
touch /mnt/data/test && ls -l /mnt/data/test
```

2. PV(서버/호스트)에서 확인

```
bash

# 서버(예: NFS)에서 숫자로 확인
ls -ln /export/myshare
# 필요 시 소유권/퍼미션 조정
chown -R 1000:1000 /export/myshare
chmod -R 0770 /export/myshare
```

3. K8s 보정

- 빠른 해결: **runAsUser: 1000, runAsGroup: 1000, fsGroup: 1000**
- 대용량/성능 고려: **fsGroupChangePolicy: OnRootMismatch** 사용
- NFS + root_squash: 서버에서 **anonuid/anongid** 또는 사전 **chown**

6) 실전 YAML 예시 (배치/서버 공통)

```
yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: myjar
  namespace: apps
spec:
  replicas: 1
  selector:
    matchLabels: { app: myjar }
  template:
    metadata:
      labels: { app: myjar }
    spec:
      securityContext:
```

```

runAsNonRoot: true
runAsUser: 1000
runAsGroup: 1000
fsGroup: 1000
fsGroupChangePolicy: OnRootMismatch
initContainers:
- name: fix-perms
  image: busybox:1.36
  command: ["sh", "-c", "chown -R 1000:1000 /data || true; chmod -R g+rwX /data || true"]
  securityContext:
    runAsUser: 0 # NFS root_squash라면 chown 실패 -> || true로 그냥 통과
    volumeMounts:
    - name: data
      mountPath: /data
containers:
- name: app
  image: myjar:1.0
  volumeMounts:
  - name: data
    mountPath: /data
volumes:
- name: data
  persistentVolumeClaim:
    claimName: myjar-pvc

```

NFS + root_squash 환경이라면 **chown**이 실패할 수 있으니 **|| true**로 넘어가고, 서버에서 미리 맞추는 게 정답에 가깝습니다.

7) 문제 터졌을 때 30초 진단 루트

1. `kubectl exec -it pod -- sh` → `id`로 실행 **UID/GID** 확인
2. `stat -c '%u:%g %a' /data`로 **PV 디렉터리 소유/퍼미션** 확인
3. 숫자 안 맞으면
 - 가장 빠름: `runAsUser/runAsGroup`을 PV 숫자에 맞춤
 - 장기: 서버에서 디렉터리 소유권/퍼미션 정렬(+ `fsGroup`)
 - NFS라면 `root_squash/anonuid/anongid` 체크

필요하시면 지금 쓰시는 **스토리지 타입(NFS/Longhorn/iSCSI/Isilon 등)** 알려주세요. 해당 드라이버 기준으로 “최소 변경으로 통과하는” 권장 세팅 바로 뽑아드릴게요.