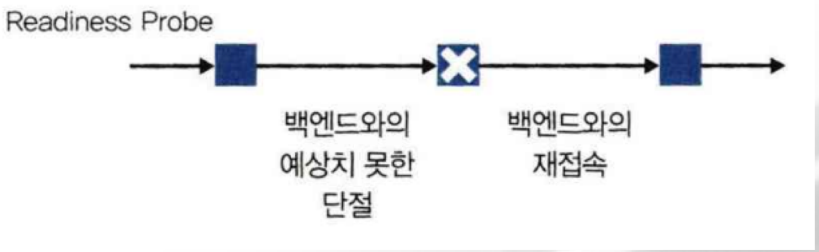
HealthCheck_LifeCycle

Health Check

- Health Check 방법
  - ✓ Liveness Probe
    - 파드 내부의 컨테이너가 정상 동작 중인지 확인하는 것으로 실패 한 경우 컨테이너를 재기동
    - Liveness Probe는 파드 내부의 컨테이너가 정상적으로 동작 중인지 확인하기 위한 헬스 체크다
    - 프로세스에 버그 등이 있을 경우 장기간 실행하고 있으면 메모리 누수(memory leak) 등에 의해 응답하지 않게 되는 경우가 있는데 이런 경우 컨테이너를 재시작할 수 있도록 Liveness Probe가 준비되어 있음
    - 헬스 체크에 한번 실패하면 재시작 없이는 복구가 어려운 컨테이너에 사용

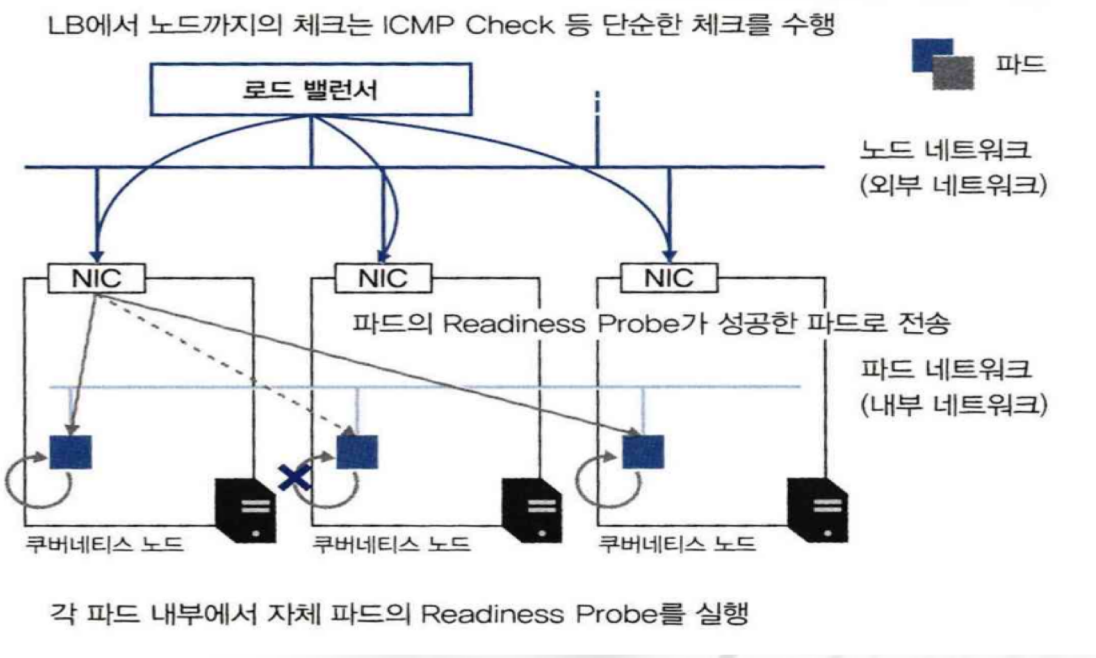
Health Check

- Health Check 방법
  - ✓ Readiness Probe
    - 파드가 요청을 받아들일 수 있는지 확인하는 것으로 실패 한 경우 트래픽을 차단하는 데 파드를 재기동하지 않음
    - Readiness Probe는 파드가 요청을 받아들일 수 있는지 확인하기 위한 헬스 체크
    - 백엔드 데이터베이스에 정상적으로 접속되는지, 캐시에 로드가 끝났는지, 기동 시간이 오래 걸리는 프로세스가 기동을 완료했는지 등을 체크
    - 파드 기동이 끝난 후에 Readiness Probe가 실패한 경우 서비스를 통한 트래픽이 파드에 전송되지 않게 하며 Liveness Probe와 달리 컨테이너를재기동하지 않음
    - 정상적으로 요청을 처리하는 수를 제어하기 위해 Readiness Probe를 실패시키는 방법도 사용할 수 있음



Health Check

- Health Check 방법
  - ✓ 로드 밸런서의 헬스 체크와 쿠버네티스 헬스 체크
    - 일반적으로 type: LoadBalancer의 서비스로 로드 밸런서를 생성한 경우 로드 밸런서에서 쿠버네티스 노드의 헬스 체크는 ICMP를 통한 단순한 체크만 하므로 파드 상태를 체크하여 트래픽을 제어하려면 Readiness Probe 또는 Liveness Probe를 제대로 설정해야 함

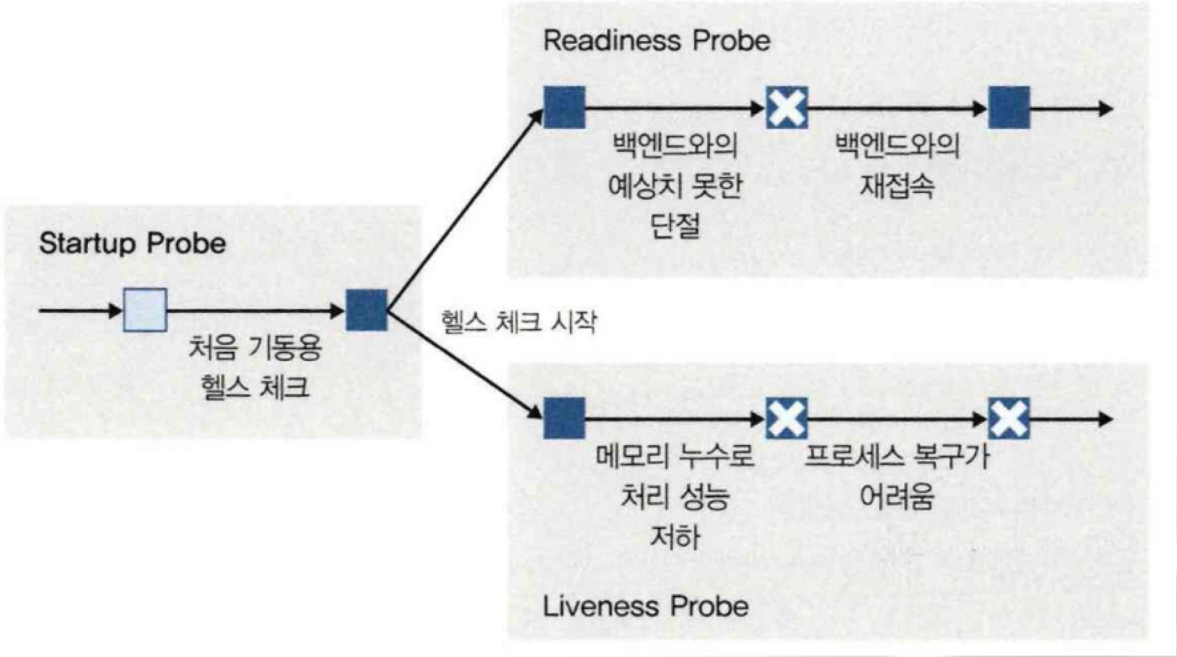


Health Check

- Health Check 방법
  - ✓ Startup Probe
    - 파드의 첫 번째 기동이 완료되었는지 확인하는 것으로 다른 Probe 실행을 시작하지 않음
    - 처음 기동하는 데 시간이 걸리는 경우 Liveness Probe를 사용하려면 첫 번째 체크 시작 시간을 연장하거나 실패라고 판단할 때까지 시간을 연장하여 사용해야 하는데 예를 들어 첫 번째 체크 시작 시간을 60초 후 등으로 설정하면 최초 기동이 바로 끝나는 경우 60초간 체크가 없는 상태가 되거나 처음 기동에 60초 이상 걸리는 경우에는 체크에 실패하는데 후자의 경우에는 몇 번이든 재기동을 반복하므로 결국 Liveness Probe 때문에 파드가 전혀 기동하지 않는 경우도 있음
    - 실패라고 판단할 때까지 시간을 연장하면 위의 문제는 발생하지 않지만 그 결과 장애 발생 시 헬스 체크가 바로 실패라고 판단해주지 않는 문제도 발생하게 되기 때문에 등장한 것이 Startup Probe
    - Startup Probe가 끝날 때까지는 Liveness Probe나 Readiness Probe가 시작되지 않으며 해당 파드가 서비스되거나 정지되지 않음

Health Check

- Health Check 방법
  - ✓ Startup Probe



Health Check

- Health Check 방식: 헬스 체크는 컨테이너 별로 이루어지며 어느 하나의 컨테이너라도 실패하면 전체 파드가 실패한 것으로 간주
  - ✓ Health Check 방식 종류
    - exec
      - 명령어를 실행하고 종료 코드가 0이 아니면 실패
      - 명령어 기반의 체크 판단 조건
        - 성공: 종료 코드가 0
        - 실패: 종료 코드가 0 이외
      - 샘플

```
livenessProbe:
  exec:
    command: ["test", "-e", "/ok.txt"]
```

Health Check

- ❖ Health Check 방식
  - ✓ Health Check 방식 종류
    - httpGet
      - ◆ HTTP GET 요청의 Status Code로 확인
      - ◆ HTTP GET 요청을 실행하고 Status Code가 200~399가 아니면 실패
      - ◆ 설정 항목으로는 path, scheme(http/https), host(httpHeaders로 지정도 가능), httpHeaders 등과 같은 세부 설정이 가능
      - ◆ HTTP GET 요청은 kubelet에서 이루어짐
      - ◆ 컨테이너에는 kubelet이 가진 파드용 네트워크 인터페이스 IP 주소로 HTTP GET 요청이 오게 됨
      - ◆ 컨테이너 내부의 애플리케이션 측에서 소스 IP 주소를 제한하는 경우에는 주의
      - ◆ 샘플

```
livenessProbe:
  httpGet:
    path: /health
    port: 80
    scheme: HTTP
    host: web.example.com
    httpHeaders:
    - name: Authorization
      value: Bearer TOKEN
```

Health Check

- Health Check 방식
  - ✓ Health Check 방식 종류
    - tcpSocket
      - TCP 세션 활성화를 검증하여 확인
      - 판단 조건
        - 성공: TCP 세션 활성화 가능
        - 실패: TCP 세션 활성화 불가능
      - 샘플

```
livenessProbe:
  tcpSocket:
    port: 80
```

  - ✓ gRPC를 사용한 애플리케이션에서 헬스 체크를 하는 경우 httpGet으로 체크하기 위해 별도 HTTP 엔드포인트를 준비하거나 tcpSocket으로 포트 활성화만 체크할 수도 있는데,이러한 방법은 헬스 체크의 정확도가 낮은데 해결 방법은 gRPC에는 원래 gRPC Health Checking Protocol을 사용한 헬스 체크 기능이 있으며 grpc_health_probe를 사용하면 엔드포인트에 대해 헬스 체크를 할 수 있는데 exec.command에서 실행하기 위해 사전에 컨테이너 이미지에 grpc_health_probe 바이너리를 추가해 두어야 함

```
livenessProbe:
  exec:
    command: ["/bin/grpc_health_probe", "-addr=:5000"]
```

Health Check

- Health Check 간격
  - ✓ Liveness Probe, Readiness Probe, Startup Probe 모두 다섯 가지 헬스 체크 간격에 대해 파라미터 설정이 가능
  - ✓ 헬스 체크 간격에 대한 설정 항목
    - initialDelaySeconds: 첫 번째 헬스 체크 시작까지의 지연(최대 faulureThreshold 만큼 연장)
    - periodSeconds: 헬스 체크 간격 시간(초)
    - timeoutSeconds: 타임아웃까지의 시간(초)
    - successThreshold: 성공이라고 판단하기까지의 체크 횟수
    - failureThreshold: 실패라고 판단하기까지의 체크 횟수
  - ✓ Liveness Probe에서는 실패했을 때 파드가 재시작하기 때문에 실패라고 판단하기까지의 체크 횟수(failureThreshold)를 환경에 맞게 설정하도록 해야 함
  - ✓ successThreshold는 1 이상이어야 하며 Liveness Probe와 Startup Probe의 경우 반드시 1이어야만 함
  - ✓ 첫 번째 체크까지 지연(initialDelaySeconds)은 가급적 사용하지말고 대신 Startup Probe를 병용해서 사용

Health Check

- Health Check 간격
  - ✓ 샘플

```
livenessProbe:
  initialDelaySeconds: 5
  periodSeconds: 5
  timeoutSeconds: 1
  successThreshold: 1
  failureThreshold: 1
```

Health Check

- Health Check 생성
  - ✓ 헬스 체크 방법 세 가지와 헬스 체크 방식 세 가지로 총 아홉 가지 패턴의 헬스 체크를 생성할 수 있음
  - ✓ Liveness Probe, Readiness Probe, Startup Probe는 하나만 지정해도 되고 모두 지정할 수도 있음

Health Check

- Health Check 생성
  - ✓ Liveness Probe로 http://localhost:80/index.html에 HTTP GET 요청을 체크하고 Readiness Probe로 /usr/share/nginx/html/50x.html 파일이 있는지 확인
    - 매니페스트: sample-healthcheck.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-healthcheck
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    livenessProbe:
      httpGet:
        path: /index.html
        port: 80
        scheme: HTTP
      timeoutSeconds: 1
      successThreshold: 1
      failureThreshold: 2
      initialDelaySeconds: 5
      periodSeconds: 3
```

Health Check

- Health Check 생성
  - ✓ Liveness Probe로 http://localhost:80/index.html에 HTTP GET 요청을 체크하고 Readiness Probe로 /usr/share/nginx/html/50x.html 파일이 있는지 확인
    - 매니페스트: sample-healthcheck.yaml

```
readinessProbe:
  exec:
    command: ["ls", "/usr/share/nginx/html/50x.html"]
  timeoutSeconds: 1
  successThreshold: 2
  failureThreshold: 1
  initialDelaySeconds: 5
  periodSeconds: 3
```

Health Check

- Health Check 생성
  - ✓ Liveness Probe로 http://localhost:80/index.html에 HTTP GET 요청을 체크하고 Readiness Probe로 /usr/share/nginx/html/50x.html 파일이 있는지 확인
    - 리소스 생성: kubectl apply -f sample-healthcheck.yaml
    - 파드에 설정된 Probe 확인: kubectl describe pod sample-healthcheck | egrep -E "Liveness|Readiness"

      Liveness: http-get http://:80/index.html delay=5s timeout=1s period=3s #success=1 #failure=2

      Readiness: exec [ls /usr/share/nginx/html/50x.html] delay=5s timeout=1s period=3s #success=2 #failure=1

Health Check

- ❖ Liveness Probe 실패
  - ✓ http://localhost:80/index.html에 httpGet이 설정되어 있어 HTTP GET 요청에 대해 200~399 Status Code가 반환되는지를 체크
    - ● 매니페스트: sample-liveness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-liveness
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    livenessProbe:
      httpGet:
        path: /index.html
        port: 80
        scheme: HTTP
      timeoutSeconds: 1
      successThreshold: 1
      failureThreshold: 2
      initialDelaySeconds: 5
      periodSeconds: 3
```

Health Check

- Liveness Probe 실패
  - ✓ http://localhost:80/index.html에 httpGet이 설정되어 있어 HTTP GET 요청에 대해 200~399 Status Code가 반환되는지를 체크
    - 리소스 생성: kubectl apply -f sample-liveness.yaml
    - 파드 확인: kubectl get pods --watch

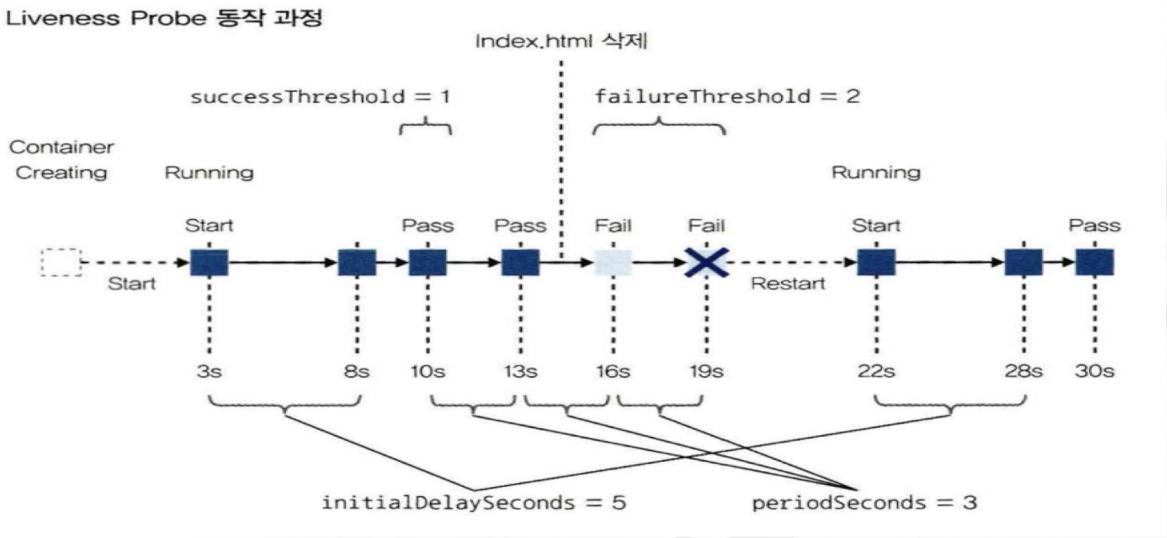
```
NAME              READY   STATUS    RESTARTS   AGE
sample-liveness   1/1     Running   0          23s
```

    - 헬스 체크 파일 삭제: kubectl exec -it sample-liveness -- rm -f /usr/share/nginx/html/index.html
    - 파드 확인: kubectl get pods --watch

```
NAME              READY   STATUS    RESTARTS      AGE
sample-liveness   1/1     Running   0             23s
sample-liveness   1/1     Running   1 (0s ago)    2m22s
sample-liveness   1/1     Running   2 (0s ago)    4m1s
```

Health Check

- ❖ Liveness Probe 실패
  - ✓ http://localhost:80/index.html에 httpGet이 설정되어 있어 HTTP GET 요청에 대해 200~399 Status Code가 반환되는지를 체크
    - 파드 확인: kubectl get pods –watch
      - ◆ Liveness Probe가 실패한 경우 파드의 restartPolicy(기본값은 Always)에 따라 컨테이너를 재시작하기 때문에 RESTARTS 카운트가 증가하는 것을 확인할 수 있음
      - ◆ 처음 5s 지연 후 10s에 1회 헬스 체크가 성공하여 파드는 Ready 상태로 변경됨
      - ◆ 13~ 16s 사이에 index.html 파일이 삭제된 후 16s 와 19s에 2회 헬스 체크가 실패하여 파드가 재시작 됨
      - ◆ 8s~ 10s와 28~30s 사이는 시스템상의 최초 지연으로 간주



Health Check

- ❖ Liveness Probe 실패
  - ✓ http://localhost:80/index.html에 httpGet이 설정되어 있어 HTTP GET 요청에 대해 200~399 Status Code가 반환되는지를 체크
    - ● 헬스 체크 상세 이력 확인: kubectl describe pod sample-liveness

```
Events:
  Type     Reason     Age                   From               Message
  ----     ------     ----                  ----               -------
  Normal   Scheduled  7m42s                 default-scheduler  Successfully assigned default/sample-liveness to worker1
  Normal   Pulled     3m42s (x3 over 7m42s)  kubelet            Container image "nginx:1.17" already present on machine
  Normal   Created    3m42s (x3 over 7m42s)  kubelet            Created container: nginx-container
  Normal   Started    3m42s (x3 over 7m42s)  kubelet            Started container nginx-container
  Warning  Unhealthy  3m42s (x4 over 5m24s)  kubelet            Liveness probe failed: HTTP probe failed with statuscode: 404
  Normal   Killing    3m42s (x2 over 5m21s)  kubelet            Container nginx-container failed liveness probe, will be restarted
```

Health Check

- ❖ Readiness Probe 실패
  - ✓ /usr/share/nginx/html/50x.html 체크
    - ● 매니페스트: sample-readiness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-readiness
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    readinessProbe:
      exec:
        command: ["ls", "/usr/share/nginx/html/50x.html"]
      timeoutSeconds: 1
      successThreshold: 2
      failureThreshold: 1
      initialDelaySeconds: 5
      periodSeconds: 3
```

Health Check

- ❖ Readiness Probe 실패
  - ✓ /usr/share/nginx/html/50x.html 체크
    - 리소스 생성: kubectl apply -f sample-readiness.yaml
    - 리소스 확인: kubectl get pods sample-readiness --watch

```
NAME                READY   STATUS    RESTARTS   AGE
sample-readiness    1/1     Running   0          96s
```

Health Check

- ❖ Readiness Probe 실패
  - ✓ /usr/share/nginx/html/50x.html 체크
    - 감시하고 있는 파일 삭제: kubectl exec -it sample-readiness -- rm -f /usr/share/nginx/html/50x.html
    - 리소스 확인: kubectl get pods sample-readiness --watch

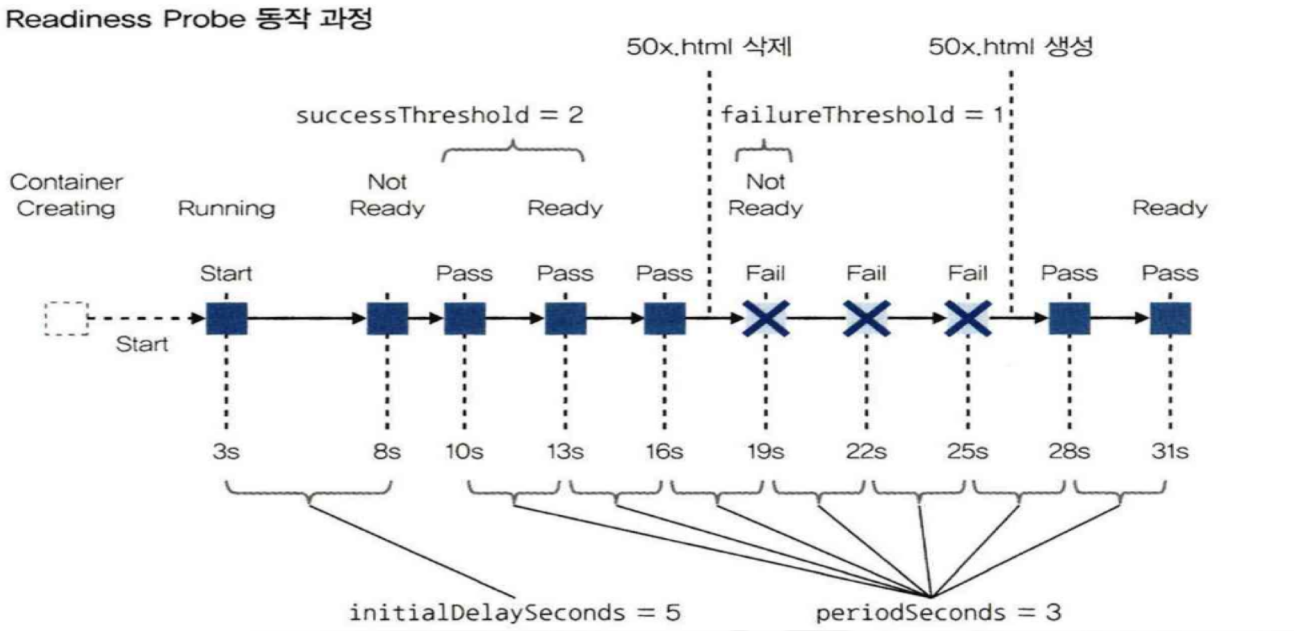
```
NAME                READY   STATUS    RESTARTS   AGE
sample-readiness    1/1     Running   0          96s
sample-readiness    0/1     Running   0          3m4s
```

    - 감시하고 있는 파일 생성: kubectl exec -it sample-readiness -- touch /usr/share/nginx/html/50x.html
    - 리소스 확인: kubectl get pods sample-readiness --watch

```
NAME                READY   STATUS    RESTARTS   AGE
sample-readiness    1/1     Running   0          96s
sample-readiness    0/1     Running   0          3m4s
sample-readiness    1/1     Running   0          4m16s
```

Health Check

- ❖ Readiness Probe 실패
  - ✓ /usr/share/nginx/html/50x.html 체크
    - Readiness Probe가 실패한 경우 READY 상태가 아니므로 서비스에서 트래픽 전송을 차단하고 Liveness Probe와는 달리 컨테이너가 재시작되지 않음
    - Readiness Probe 동작 과정을 그림으로 표현
    - 처음 5s 지연 후 10s와 13s에 2회 헬스 체크가 성공하여 파드가 Ready 상태로 바뀌고 그 후 16~ 19s 사이에 50x.html을 삭제한 후 1회 헬스 체크가 실패하고 파드는 Not Ready 상태가 되며 25~28s 사이에 50x.html을 생성하면 다시 Ready 상태가 됨



Health Check

- ❖ Readiness Probe 실패
  - ✓ /usr/share/nginx/html/50x.html 체크에서 Events에서 헬스 체크 상세 이력을 확인
    - ● 파드 이력 정보: kubectl describe pod sample-readiness

```
Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  8m34s                default-scheduler  Successfully assigned default/sample-readiness to worker2
  Normal   Pulled     8m34s                kubelet            Container image "nginx:1.17" already present on machine
  Normal   Created    8m34s                kubelet            Created container: nginx-container
  Normal   Started    8m34s                kubelet            Started container nginx-container
  Warning  Unhealthy  4m32s (x22 over 5m32s)  kubelet            Readiness probe failed: ls: cannot access '/usr/share/nginx/html/50x.html': No such file or directory
```

Health Check

- ❖ Readiness Probe 실패
  - ✓ 추가 Ready 조건을 추가하는 파드 ready++(ReadinessGate)
    - 특정한 경우에 시스템을 전체로 봤을 때 그 파드가 정말 Ready 상태인지를 추가로 체크하고 싶은 때가 있는데 그 기능을 제공하는 것이 파드 ready++ 로 이 기능은 클라우드 외부 로드 밸런서 와의 연계에 시간이 걸리는 경우나 일반적인 파드의 Ready 판단만으로는 롤링 업데이트 시 기존 파드가 한 번에 사라져 안전하게 업데이트할 수 없는 경우 등에도 사용(GKE의 Container-native 로드 밸런서, EKS의 ALB 인그레스 등)
    - 파드 ready++는 그 파드에 대해 ReadinessGate라는 추가 체크 항목(spec.readinessGates)을 설정
    - ReadinessGate를 통과할 때까지 서비스 전송 대상에 추가되지 않고 롤링 업데이트 시 다음 파드 기동으로 이동하지 않음

Health Check

- ❖ Readiness Probe 실패
  - ✓ 추가 Ready 조건을 추가하는 파드 ready++(ReadinessGate)
    - ● 매니페스트: sample-readinessgate.yaml

```
---
apiVersion: v1
kind: Pod
metadata:
  name: sample-readinessgate
  labels:
    app: sample-readinessgate
spec:
  readinessGates:
  - conditionType: "amsy.dev/sample-condition"
  containers:
  - name: nginx-container
    image: nginx:1.17
```

Health Check

- ❖ Readiness Probe 실패
  - ✓ 추가 Ready 조건을 추가하는 파드 ready++(ReadinessGate)
    - ● 매니페스트: sample-readinessgate.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  name: sample-readinessgate
spec:
  type: ClusterIP
  ports:
  - name: "http-port"
    protocol: "TCP"
    port: 8080
    targetPort: 80
  selector:
    app: sample-readinessgate
```

Health Check

- ❖ Readiness Probe 실패
  - ✓ 추가 Ready 조건을 추가하는 파드 ready++(ReadinessGate)
    - ● ReadinessGate는 여러 설정이 가능한데 이 경우는 모든 상태가 Ready가 되지 않으면 파드는 Ready 상태가 되지 않음

```
spec:
  readinessGates:
  - conditionType: "amsy.dev/sample-condition"
  - conditionType: "amsy.dev/sample-condition2"
```

    - ● 리소스 생성: kubectl apply -f sample-readinessgate.yaml
      - ◆ ReadinessGate가 설정된 파드를 기동하면 READINESS GATES 항목에서 추가 상태가 표시됨
      - ◆ 기동하고 있는 상태에서도 서비스에 전송 대상이 등록되어 있지 않지만 API 서버에 직접 상태 변경을 요청하고 정보를 업데이트하면 서비스에 전송 대상이 등록된 것을 확인할 수 있음

Health Check

- ❖ Readiness Probe 실패
  - ✓ 추가 Ready 조건을 추가하는 파드 ready++(ReadinessGate)
    - 파드의 ReadinessGate 상태를 확인: kubectl get pod sample-readinessgate -o wide

```
NAME                    READY   STATUS    RESTARTS   AGE     IP           NODE
NOMINATED NODE   READINESS GATES
sample-readinessgate    1/1     Running   0          4m56s   10.244.2.29  worker1
<none>           0/1
```

    - 서비스 전송 대상 확인: kubectl describe service sample-readinessgate

```
Name:              sample-readinessgate
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=sample-readinessgate
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.98.78.234
IPs:               10.98.78.234
Port:              http-port  8080/TCP
TargetPort:        80/TCP
Endpoints:
Session Affinity:  None
Events:            <none>
```

Health Check

- ❖ Readiness Probe 실패
  - ✓ 추가 Ready 조건을 추가하는 파드 ready++(ReadinessGate)
    - API 서버에 프록시를 로컬로 생성: kubectl proxy

      Starting to serve on 127.0.0.1:8001
    - ReadinessGate 상태를 업데이트: curl -X PATCH -H "Content-Type: application/json-patch+json" -d '[{"op": "add", "path": "/status/conditions/-", "value": {"type": "amsy.dev/sample-condition","status":"True"}}]' http://localhost:8001/api/v1/namespaces/default/pods/sample-readinessgate/status
    - 파드의 ReadinessGate 상태를 확인: kubectl get pod sample-readinessgate -o wide

      NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

      sample-readinessgate 1/1 Running 0 12m 10.244.2.29 worker1 <none> 1/1

Health Check

- ❖ Readiness Probe 실패
  - ✓ 추가 Ready 조건을 추가하는 파드 ready++(ReadinessGate)
    - ● 서비스 전송 대상 확인: kubectl describe service sample-readinessgate

```
Name:              sample-readinessgate
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=sample-readinessgate
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.98.78.234
IPs:               10.98.78.234
Port:              http-port  8080/TCP
TargetPort:        80/TCP
Endpoints:         10.244.2.29:80
Session Affinity:  None
Events:            <none>
```

Health Check

- ❖ Readiness Probe 실패
  - ✓ ReadinessGate가 설정된 디플로이먼트 등을 업데이트하는 경우 롤링 업데이트에서는 ReadinessGate가 Ready가 되지 않으면 다음 파드 변경으로 진행하지 않음
    - 매니페스트: sample-deployment-readinessgate.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment-readinessgate
spec:
  replicas: 2
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      readinessGates:
      - conditionType: "amsy.dev/sample-condition"
      containers:
      - name: nginx-container
        image: nginx:1.17
```

Health Check

- ❖ Readiness Probe 실패
  - ✓ ReadinessGate가 설정된 디플로이먼트 등을 업데이트하는 경우 롤링 업데이트에서는 ReadinessGate가 Ready가 되지 않으면 다음 파드 변경으로 진행하지 않음
    - 리소스 생성: kubectl apply -f sample-deployment-readinessgate.yaml
    - 이미지 업데이트: kubectl set image deployment sample-deployment-readinessgate nginx-container=nginx
    - 파드 확인: kubectl get pods -o wide

```
NAME                                                READY   STATUS    RESTARTS   AGE
IP              NODE      NOMINATED NODE   READINESS GATES
sample-deployment-readinessgate-55597fd666-mdfxc    1/1     Running   0
44s   10.244.1.26   worker2   <none>           0/1
sample-deployment-readinessgate-bdb494b4f-9zz9t     1/1     Running   0
89s   10.244.1.25   worker2   <none>           0/1
sample-deployment-readinessgate-bdb494b4f-wrmhd     1/1     Running   0
89s   10.244.2.30   worker1   <none>           0/1
```

Health Check

- Readiness Probe를 무시한 서비스 생성
  - ✓ Readiness Probe가 실패하는 경우에는 서비스(엔드포인트)에서 제외된 상태가 되지만 스테이트풀셋에서 Headless 서비스를 사용할 때는 파드가 Ready 상태가 되지 않아도 클러스터를 구성하기 위해 각 파드의 이름 해석이 필요한 경우가 있음
  - ✓ Readiness Probe가 실패한 경우에도 서비스에 연결되게 하려면 spec.publishNotReadyAddresses를 true로 설정해야 함

Health Check

- Readiness Probe를 무시한 서비스 생성
  - ✓ 매니페스트: sample-publish-notready.yaml

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sample-publish-notready
spec:
  serviceName: sample-publish-notready
  replicas: 3
  podManagementPolicy: Parallel
  selector:
    matchLabels:
      app: publish-notready
  template:
    metadata:
      labels:
        app: publish-notready
    spec:
      containers:
      - name: nginx-container
        image: amsy810/echo-nginx:v2.0
        readinessProbe:
          exec:
            command: ["sh", "-c", "exit 1"]
```

Health Check

- Readiness Probe를 무시한 서비스 생성
  - ✓ 매니페스트: sample-publish-notready.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  name: sample-publish-notready
spec:
  type: ClusterIP
  publishNotReadyAddresses: true
  ports:
  - name: "http-port"
    protocol: "TCP"
    port: 8080
    targetPort: 80
  selector:
    app: publish-notready
```

Health Check

- Readiness Probe를 무시한 서비스 생성
  - ✓ 리소스 배포: kubectl apply -f sample-publish-notready.yaml
  - ✓ spec.publishNotReadyAddresses가 true로 된 경우 파드가 Ready 상태가 아니어도 이름 해석이 가능한데 이는 로드 밸런서에도 포함되어 버리기 때문에 특별한 경우에만 사용
  - ✓ 서비스를 생성하고 나서도 중간에 변경할 수 있음
  - ✓ 파드 확인: kubectl get pods -o wide

```
NAME                       READY   STATUS             RESTARTS      AGE   IP
NODE      NOMINATED NODE   READINESS GATES
sample-publish-notready-0  0/1     Running   3 (18s ago)   70s   10.244.2.31   worker1
<none>           <none>
sample-publish-notready-1  0/1 Running   3 (14s ago)   70s   10.244.3.25   worker3
<none>           <none>
sample-publish-notready-2  0/1 Running   3 (15s ago)   70s   10.244.1.29   worker2
<none>           <none>
```

Health Check

- Readiness Probe를 무시한 서비스 생성
  - ✓ 서비스와 연결된 EndPoint 확인: kubectl get endpoints sample-publish-notready

    NAME ENDPOINTS AGE

    sample-publish-notready 10.244.1.29:80,10.244.2.31:80,10.244.3.25:80 10m
  - ✓ 서비스 이름 해석 확인: kubectl run --image=amsy810/tools:v2.0 --restart=Never --rm -i testpod -- dig sample-publish-notready-0.sample-publish-notready.default.svc.cluster.local
  - ✓ publishNotReadyAddresses 비활성화: kubectl patch service sample-publish-notready --patch '{"spec": {"publishNotReadyAddresses": false}}'
  - ✓ 서비스와 연결된 EndPoint 확인: kubectl get endpoints sample-publish-notready

    NAME ENDPOINTS AGE

    sample-publish-notready 18m
  - ✓ 서비스 이름 해석 확인: kubectl run --image=amsy810/tools:v2.0 --restart=Never --rm -i testpod -- dig sample-publish-notready-0.sample-publish-notready.default.svc.cluster.local

Health Check

- ❖ Startup Probe를 사용한 지연 체크 와 실패
  - ✓ Startup Probe를 사용하여 Liveness Probe, Readiness Probe에 의한 체크를 지연시키고 /root/startup 파일이 존재하는 지를 명령어 기반으로 체크하고 명령어 실행 시간과 어느 Probe가 실행되었는지가 /root/log 파일에 기록되는 예제
    - 매니페스트: sample-startup.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-startup
spec:
  containers:
  - name: tools-container
    image: nginx
```

Health Check

- Startup Probe를 사용한 지연 체크 와 실패
  - ✓ Startup Probe를 사용하여 Liveness Probe, Readiness Probe에 의한 체크를 지연시키고 /root/startup 파일이 존재하는 지를 명령어 기반으로 체크하고 명령어 실행 시간과 어느 Probe가 실행되었는지가 /root/log 파일에 기록되는 예제
    - 매니페스트: sample-startup.yaml

```
    livenessProbe:
      exec:
        command: ["sh", "-c", "echo [$(date)] liveness >> /root/log; test ! -e
/root/liveness"]
      periodSeconds: 3
    readinessProbe:
      exec:
        command: ["sh", "-c", "echo [$(date)] readiness >> /root/log; test ! -e
/root/readiness"]
      periodSeconds: 3
    startupProbe:
      exec:
        command: ["sh", "-c", "echo [$(date)] startup >> /root/log; test -e
/root/startup"]
      failureThreshold: 100
      periodSeconds: 3
```

Health Check

- Startup Probe를 사용한 지연 체크 와 실패
  - ✓ Startup Probe를 사용하여 Liveness Probe, Readiness Probe에 의한 체크를 지연시키고 /root/startup 파일이 존재하는 지를 명령어 기반으로 체크하고 명령어 실행 시간과 어느 Probe가 실행되었는지가 /root/log 파일에 기록되는 예제
    - 리소스 배포: kubectl apply -f sample-startup.yaml
    - 리소스 확인: kubectl get pods sample-startup --watch

      NAME READY STATUS RESTARTS AGE

      sample-startup 0/1 Running 11 (119s ago) 56m
    - Startup Probe 성공: kubectl exec -it sample-startup -- touch /root/startup
    - 리소스 확인: kubectl get pods sample-startup --watch

      NAME READY STATUS RESTARTS AGE

      sample-startup 0/1 Running 11 (119s ago) 56m

      sample-startup 0/1 Running 11 (2m21s ago) 57m

      sample-startup 1/1 Running 11 (2m21s ago) 57m

Health Check

- ❖ Startup Probe를 사용한 지연 체크 와 실패
  - ✓ Startup Probe를 사용하여 Liveness Probe, Readiness Probe에 의한 체크를 지연시키고 /root/startup 파일이 존재하는 지를 명령어 기반으로 체크하고 명령어 실행 시간과 어느 Probe가 실행되었는지가 /root/log 파일에 기록되는 예제
    - 로그 확인: kubectl exec -it sample-startup -- cat /root/log
    - 리소스 확인: kubectl get pods sample-startup --watch

      NAME READY STATUS RESTARTS AGE

      sample-startup 0/1 Running 11 (119s ago) 56m
    - Startup Probe 성공: kubectl exec -it sample-startup -- touch /root/startup
    - 리소스 확인: kubectl get pods sample-startup --watch

      [Mon Feb 23 01:57:54 UTC 2026] startup

      [Mon Feb 23 01:57:57 UTC 2026] startup

      [Mon Feb 23 01:58:00 UTC 2026] startup

      [Mon Feb 23 01:58:03 UTC 2026] startup

      [Mon Feb 23 01:58:03 UTC 2026] readiness

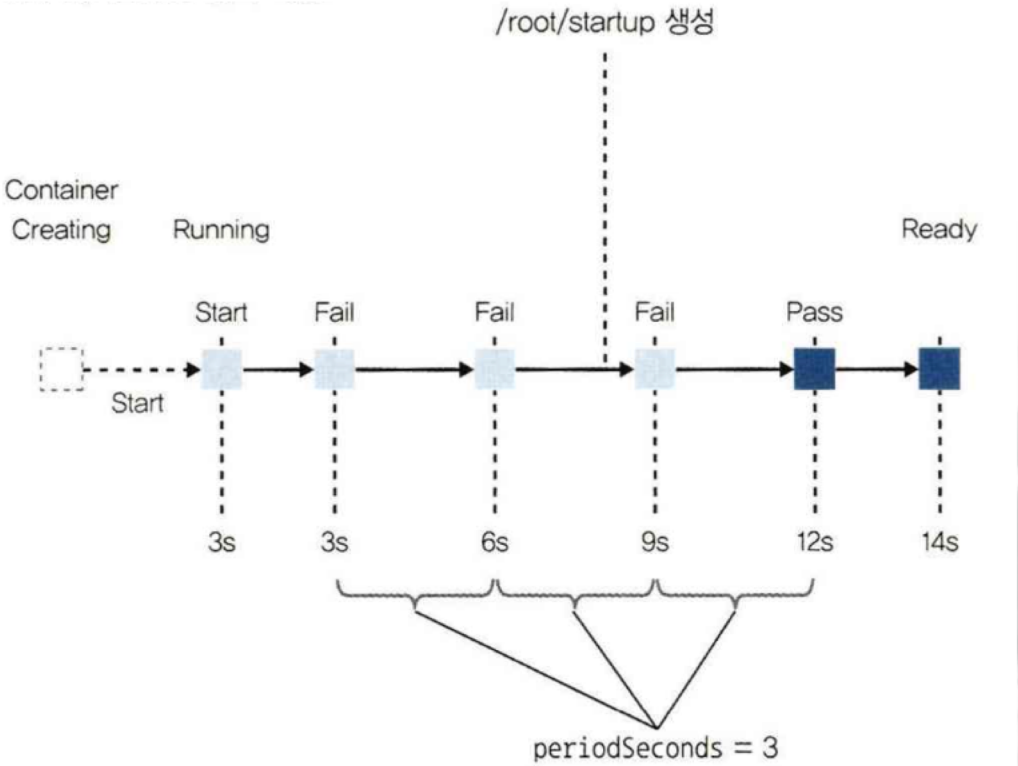
      [Mon Feb 23 01:58:06 UTC 2026] liveness

      [Mon Feb 23 01:58:06 UTC 2026] readiness

      [Mon Feb 23 01:58:09 UTC 2026] liveness

Health Check

- ❖ Startup Probe를 사용한 지연 체크 와 실패
  - ✓ Startup Probe를 사용하여 Liveness Probe, Readiness Probe에 의한 체크를 지연시키고 /root/startup 파일이 존재하는 지를 명령어 기반으로 체크하고 명령어 실행 시간과 어느 Probe가 실행되었는지가 /root/log 파일에 기록되는 예제
    - Startup Probe 동작 과정



Health Check

- Startup Probe 파라미터 사용
  - ✓ 매니페스트: sample-startup-shortfail.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-startup-shortfail
spec:
  containers:
  - name: tools-container
    image: nginx
    readinessProbe:
      exec:
        command: ["sh", "-c", "echo [$(date)] readiness >> /root/log; test ! -e
/root/readiness"]
      periodSeconds: 3
    startupProbe:
      exec:
        command: ["sh", "-c", "echo [$(date)] startup >> /root/log; test -e /root/startup"]
      failureThreshold: 3
      initialDelaySeconds: 5
      periodSeconds: 3
```

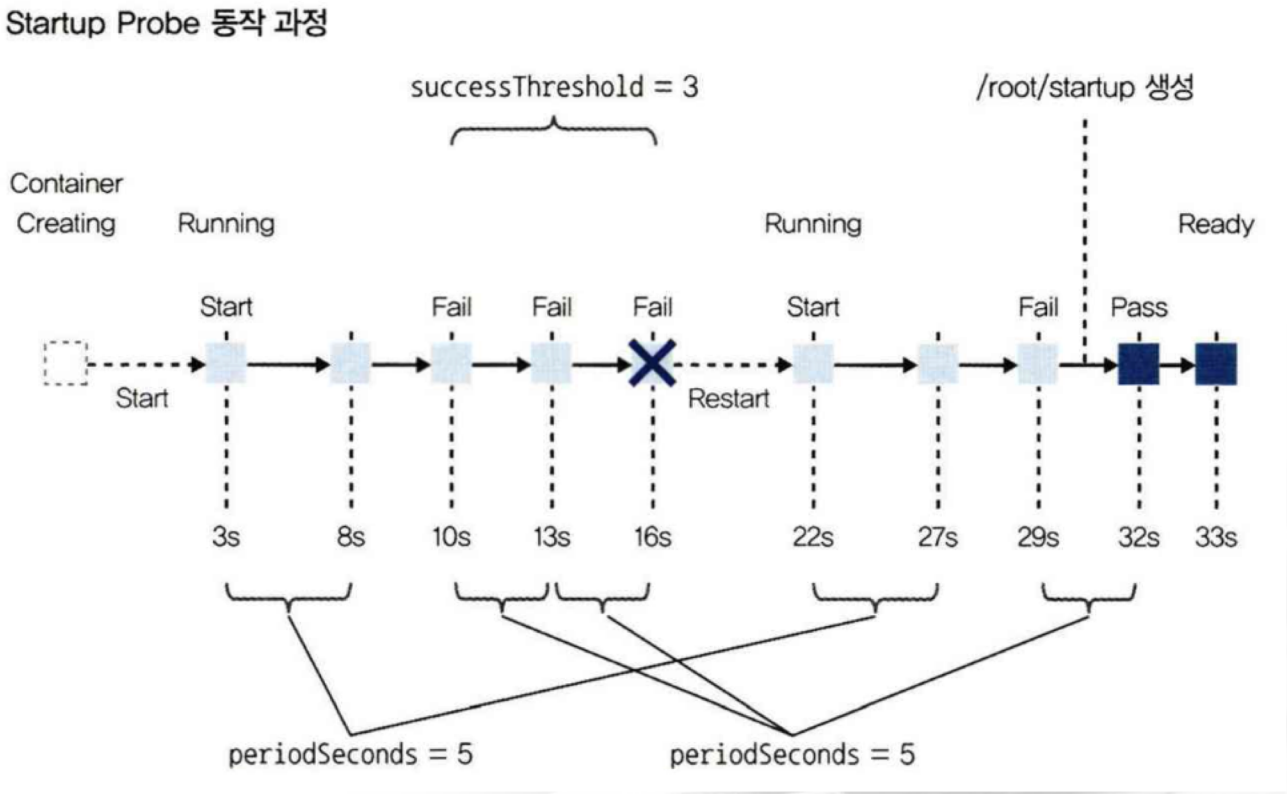
Health Check

- Startup Probe 파라미터 사용
  - ✓ 매니페스트: sample-startup-shortfail.yaml
    - Startup Probe의 경우 failureThreshold 횟수만큼 실패한 경우에는 Liveness Probe와 마찬가지로 파드가 정지되고 restartPolicy에 따라 재기동하기 때문에 Startup Probe를 사용할 때는 failureThreshold 값을 충분히 크게 두는 것이 좋음
    - 이번에는 아주 작게 3으로 설정하고 3회 체크에 실패하면 파드가 재기동되도록 설정
  - ✓ 리소스 배포: kubectl apply -fsample-startup-shortfail.yaml
  - ✓ 파일 생성
    - kubectl exec -it sample-startup-shortfail -- touch /root/startup
    - kubectl exec -it sample-startup-shortfail -- touch /root/readiness
  - ✓ 리소스 확인: kubectl get pods –watch

```
NAME                       READY   STATUS             RESTARTS      AGE
sample-startup-shortfail   0/1     Running            0             5s
sample-startup-shortfail   0/1     Running            1 (2s ago)    17s
sample-startup-shortfail   0/1     Running            2 (2s ago)    32s
sample-startup-shortfail   0/1     Running            3 (2s ago)    47s
sample-startup-shortfail   0/1     CrashLoopBackOff   3 (0s ago)    60s
sample-startup-shortfail   0/1     Running            4 (30s ago)   90s
sample-startup-shortfail   0/1     Running            4 (39s ago)   99s
sample-startup-shortfail   1/1     Running            4 (39s ago)   99s
```

Health Check

- Startup Probe 파라미터 사용
  - ✓ Startup Probe가 실패했을 때의 동작 과정



Container 정지 시 동작

- 컨테이너 정지 시 동작
  - ✓ 컨테이너 프로세스 정지 또는 헬스 체크 실패 시 컨테이너를 재기동할지는 spec.restartPolicy로 지정
  - ✓ 워크로드 리소스의 잡은 파드 한 개당 1회 명령어 실행을 성공시키는 것이 목적이므로 Always는 선택할 수 없음
  - ✓ restartPolicy
    - Always: 파드가 정지하면 항상 재가동
    - OnFailure: 예상치 못하게 파드가 정지한 경우(종료 코드 0 이외) 파드 재가동
    - Never: 파드가 정지해도 파드를 재기동하지 않음

Container 정지 시 동작

- Always: restartPolicy가 Always인 경우 파드에서 실행되는 명령어의 성공/실패에 상관없이 항상 재기동
  - ✓ 매니페스트: sample-restart-always.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-restart-always
spec:
  restartPolicy: Always
  containers:
  - name: nginx-container
    image: nginx:1.17
    command: ["sh", "-c", "exit 0"] # 정상 종료
    # command: ["sh", "-c", "exit 1"] # 비정상 종료
```

  - ✓ 리소스 생성: kubectl apply -f sample-restart-always.yaml

Container 정지 시 동작

- Always: restartPolicy가 Always인 경우 파드에서 실행되는 명령어의 성공/실패에 상관없이 항상 재기동
  - ✓ 파드 확인: kubectl get pods --watch(첫번째 Command)

```
sample-restart-always   0/1   Completed   1 (3s ago)   3s
sample-restart-always   0/1   CrashLoopBackOff   1 (15s ago)   16s
sample-restart-always   0/1   Completed   2 (16s ago)   17s
sample-restart-always   0/1   CrashLoopBackOff   2 (14s ago)   30s
sample-restart-always   0/1   Completed   3 (28s ago)   44s
sample-restart-always   0/1   CrashLoopBackOff   3 (13s ago)   56s
```

  - ✓ 파드 확인: kubectl get pods --watch(두번째 Command)

```
NAME                    READY   STATUS    RESTARTS   AGE
sample-restart-always   0/1     Pending   0          0s
sample-restart-always   0/1     Pending   0          0s
sample-restart-always   0/1     ContainerCreating   0          0s
sample-restart-always   0/1     Error     0          1s
sample-restart-always   0/1     Error     1 (2s ago)   2s
sample-restart-always   0/1     CrashLoopBackOff   1 (16s ago)   17s
sample-restart-always   0/1     Error     2 (17s ago)   18s
```

Container 정지 시 동작

- OnFailure: 파드에서 실행되는 명령어가 실패하면 재기동
  - ✓ 매니페스트: sample-restart-onfailure.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-restart-always
spec:
  restartPolicy: OnFailure
  containers:
  - name: nginx-container
    image: nginx:1.17
    command: ["sh", "-c", "exit 0"] # 정상 종료
    # command: ["sh", "-c", "exit 1"] # 비정상 종료
```

  - ✓ 리소스 생성: kubectl apply -f sample-restart-onfailure.yaml

Container 정지 시 동작

- OnFailure: 파드에서 실행되는 명령어가 실패하면 재기동
  - ✓ 파드 확인: kubectl get pods --watch(명령 성공)

```
NAME                        READY   STATUS      RESTARTS   AGE
sample-restart-onfailure    0/1     Completed   0          80s
```

  - ✓ 파드 확인: kubectl get pods --watch(명령 실패)

```
NAME                        READY   STATUS              RESTARTS      AGE
sample-restart-onfailure    0/1     Pending             0             0s
sample-restart-onfailure    0/1     Pending             0             0s
sample-restart-onfailure    0/1     ContainerCreating   0             0s
sample-restart-onfailure    0/1     Error               0             0s
sample-restart-onfailure    0/1     Error               1 (1s ago)    1s
```

Container 정지 시 동작

- Never: 파드가 종료되더라도 파드를 재기동하지 않음
  - ✓ 매니페스트: sample-restart-never.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-restart-never
spec:
  restartPolicy: Never
  containers:
  - name: nginx-container
    image: nginx:1.17
    command: ["sh", "-c", "exit 0"] # 성공
    # command: ["sh", "-c", "exit 1"] # 실패
```

  - ✓ 리소스 생성: kubectl apply -f sample-restart-never.yaml

Container 정지 시 동작

- Never: 파드가 종료되더라도 파드를 재기동하지 않음
  - ✓ 파드 확인: kubectl get pods --watch(명령 성공)

```
NAME                    READY   STATUS              RESTARTS   AGE
sample-restart-never    0/1     Pending             0          0s
sample-restart-never    0/1     ContainerCreating   0          0s
sample-restart-never    0/1     Completed           0          3s
```

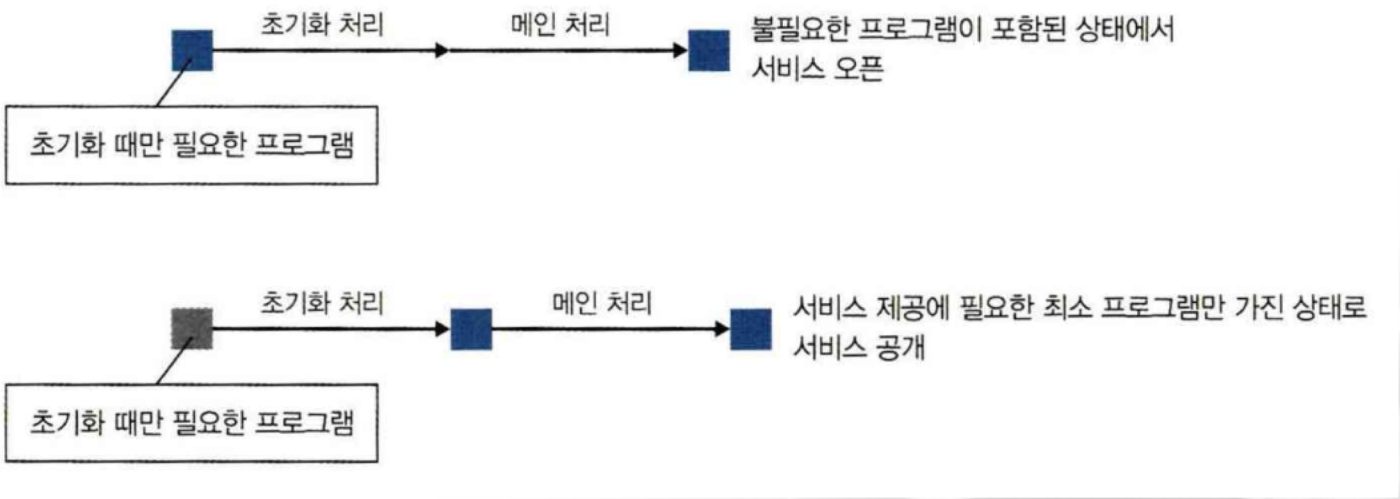
  - ✓ 파드 확인: kubectl get pods --watch(명령 실패)

```
NAME                    READY   STATUS              RESTARTS   AGE
sample-restart-never    0/1     Pending             0          0s
sample-restart-never    0/1     ContainerCreating   0          0s
sample-restart-never    0/1     Error               0          1s
```

초기화 컨테이너

❖ 개요

- ✓ 초기화 컨테이너(Init Container)란 파드 내부에서 메인이 되는 컨테이너를 기동하기 전에 별도의 컨테이너를 기동하기 위한 기능을 의미
- ✓ 초기화 컨테이너를 사용하면, 설정에 필요한 스크립트 등을 메인 컨테이너에 보관하지 않는 상태를 유지할 수 있음(보안을 유지하고 이미지 용량을 줄이기 위해)



초기화 컨테이너

- 개요
  - ✓ spec.initContainers는 복수로 지정할 수 있으며 위에서부터 하나씩 컨테이너가 기동되고 spec.initContainers[ ].command가 실행됨
  - ✓ spec.containers는 동시에 병렬로 컨테이너가 기동되기 때문에 순서가 필요한 처리 등에는 사용하기 힘들지만 초기화 컨테이너를 사용하면 가능
  - ✓ 저장소에서 파일 등을 가져오는 처리, 컨테이너 기동을 지연시키는 처리, 설정 파일을 동적으로 생성하는 처리, 서비스가 생성되어 있는지 확인하는 작업과 그 외 메인 컨테이너를 기동하기 전의 체크 작업 등에 사용

초기화 컨테이너

- 2개의 컨테이너를 이용하여 메인 nginx-container의 index.html을 생성
  - ✓ 매니페스트: sample-initcontainer.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-initcontainer
spec:
  initContainers:
  - name: output-1
    image: alpine:latest
    command: ['sh', '-c', 'sleep 20; echo 1st > /usr/share/nginx/html/index.html']
    volumeMounts:
    - name: html-volume
      mountPath: /usr/share/nginx/html/
  - name: output-2
    image: alpine:latest
    command: ['sh', '-c', 'sleep 10; echo 2nd >> /usr/share/nginx/html/index.html']
    volumeMounts:
    - name: html-volume
      mountPath: /usr/share/nginx/html/
```

초기화 컨테이너

❖ 2개의 컨테이너를 이용하여 메인 nginx-container의 index.html을 생성

✓ 매니페스트: sample-initcontainer.yaml

```
containers:
- name: nginx-container
  image: nginx:1.16
  volumeMounts:
  - name: html-volume
    mountPath: /usr/share/nginx/html/
volumes:
- name: html-volume
  emptyDir: {}
```

✓ 리소스 배포: kubectl apply -f sample-initcontainer.yaml

✓ 파드의 상태 확인: kubectl get pods --watch

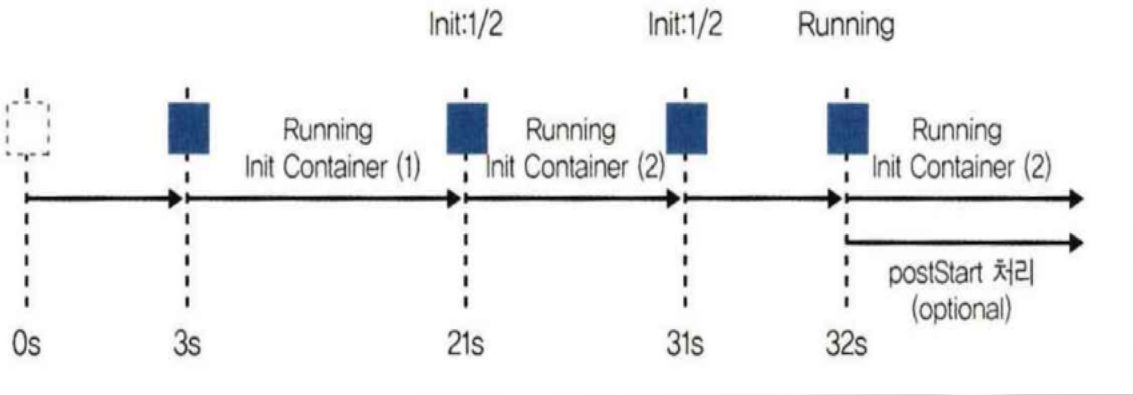
```
NAME                  READY   STATUS            RESTARTS   AGE
sample-initcontainer  0/1     Pending           0          0s
sample-initcontainer  0/1     Pending           0          0s
sample-initcontainer  0/1     Init:0/2          0          0s
sample-initcontainer  0/1     Init:0/2          0          8s
sample-initcontainer  0/1     Init:1/2          0          28s
sample-initcontainer  0/1     Init:1/2          0          30s
sample-initcontainer  0/1     PodInitializing   0          40s
sample-initcontainer  1/1     Running           0          54s
```

초기화 컨테이너

- 2개의 컨테이너를 이용하여 메인 nginx-container의 index.html을 생성
  - ✓ 초기화 컨테이너에서 추가된 파일 확인: kubectl exec -it sample-initcontainer -- cat /usr/share/nginx/html/index.html

    1st

    2nd
  - ✓ 초기화 컨테이너 동작 과정



postStart/preStop

- 컨테이너 기동 후와 정지 직전에 임의의 명령어를 실행
  - ✓ 매니페스트: sample-lifecycle-exec.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-lifecycle-exec
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    command: ["/bin/sh", "-c", "touch /tmp/started; sleep 3600"]
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "sleep 20; touch /tmp/poststart"]
      preStop:
        exec:
          command: ["/bin/sh", "-c", "touch /tmp/prestop; sleep 20"]
```

postStart/preStop

- 컨테이너 기동 후와 정지 직전에 임의의 명령어를 실행
  - ✓ 파드 생성: kubectl apply -f sample-lifecycle-exec.yaml
  - ✓ 기동 후 바로 확인: kubectl exec -it sample-lifecycle-exec -- ls /tmp

  started
  - ✓ 기동 후 20초 경과 후 확인: kubectl exec -it sample-lifecycle-exec -- ls /tmp

  poststart started
  - ✓ 파드 정지: kubectl delete -f sample-lifecycle-exec.yaml
  - ✓ 삭제 후 바로 확인: kubectl exec -it sample-lifecycle-exec -- ls /tmp

  poststart prestop started
  - ✓ 결과 해석
    - postStart는 spec.containers[].command(Entrypoint) 실행과 거의 같은 타이밍에 실행되는데 비동기로 실행되기 때문에 Entrypoint가 실행되기 전에 postStart가 실행될 가능성도 있어서 기동 시에 필요한 파일을 배치하는 등의 작업은 초기화 컨테이너를 사용하거나 Entrypoint 안에서 실행한 후에 기동하길 권장
    - preStop은 종료 요청이 온 후 실행
    - postStart와 preStop에는 명령어 실행(exec) 외에 httpGet도 사용할 수 있음

postStart/preStop

- 컨테이너 기동 직후에 HTTP 요청을 전송하는 파드
  - ✓ 매니페스트: sample-lifecycle-httpget.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: poststart-http-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: poststart-demo
```

postStart/preStop

- 컨테이너 기동 직후에 HTTP 요청을 전송하는 파드
  - ✓ 매니페스트: sample-lifecycle-httpget.yaml

```
template:
  metadata:
    labels:
      app: poststart-demo
  spec:
    containers:
    - name: main-container
      image: nginx:alpine  # 테스트를 위해 80포트가 열려있는 nginx 사용
      ports:
      - containerPort: 80
      lifecycle:
        postStart:
          httpGet:
            path: /        # 호출할 경로
            port: 80       # 호출할 포트
            host: localhost # 기본값은 Pod IP입니다.
            scheme: HTTP  # HTTP 또는 HTTPS
      resources:
        limits:
          cpu: "100m"
          memory: "64Mi"
```

postStart/preStop

- 주의사항
  - ✓ postStart와 preStop은 최소 한 번 실행되는데 여러 번 실행될 가능성도 있으므로 한 번만 실행이 허용되는 처리를 실행할 때는 주의
  - ✓ postStart에 타임아웃 설정을 할 수 없다는 점도 주의해야 하는데 postStart 실행 중에는 Probe도 실행되지 않아 postStart 실행에 시간이 걸리는 경우에도 LivenessProbe나 StartupProbe로 실패시킬 수 없기 때문에 postStart에서 교착 상태(deadlock)로 끝나지 않는 프로그램 등은 실행하지 않도록 해야하는데 그런 프로그램을 실행할 경우에는 프로그램 측에 타임아웃 설정을 넣어야 함
  - ✓ 재시도 메커니즘 없는데 만약 httpGet 요청이 실패하면(HTTP 상태 코드가 200-399가 아니면) 쿠버네티스는 컨테이너를 실패(Failed)로 간주하고 종료(kill)시킨 뒤 재시작 정책에 따라 다시 뛰움
  - ✓ Liveness/Readiness Probe와의 차이:
    - Probes: 컨테이너가 실행되는 동안 주기적으로 상태를 체크
    - postStart: 컨테이너 시작 시 딱 한 번만 실행
  - ✓ 사용하는 경우
    - 초기화 신호: 컨테이너가 시작되었음을 외부 모니터링 시스템이나 로드밸런서에 알릴 때
    - 데이터 워밍업: 앱이 시작되자마자 특정 캐시를 채우도록 내부 API를 호출할 때

리소스 삭제

- 리소스 삭제
  - ✓ 쿠버네티스에서는 레플리카셋 등의 상위 리소스가 삭제되면 하위 리소스가 되는 파드 등을 삭제하기 위해 가비지 수집(Garbage Collection)을 수행
  - ✓ 레플리카셋 등에서는 파드를 생성하지만 생성된 파드에는 어떤 레플리카셋으로 생성되었는지를 판별하기 위해 metadata.ownerReferences 아래에 자동으로 정보를 저장하는데 이것은 디플로이먼트와 레플리카셋에서도 마찬가지
  - ✓ 레플리카셋을 삭제했을 때의 동작
    - Background(기본값): 레플리카셋을 즉시 삭제하고 파드는 가비지 수집기가 백그라운드에서 비동기로 삭제
    - Foreground
      - 레플리카셋을 즉시 삭제하지 않고 deletionTimestamp, metadata.finalizers = foregroundDeletion으로 설정
      - 가비지 수집기가 각 파드에서 blockOwnerDeletion = true인 것을 삭제하는데 false인 것은 Foreground 삭제라도 Background로 삭제
      - 모든 삭제가 끝나면 레플리카셋을 삭제
    - Orphan: 레플리카셋 삭제 시 파드 삭제를 하지 않음
  - ✓ kubectl로 선택할 수 있는 옵션은 Background와 Orphan뿐이며 Foreground를 사용하려면 API를 조작해야 함

리소스 삭제

- 리소스 삭제
  - ✓ 쿠버네티스에서는 레플리카셋 등의 상위 리소스가 삭제되면 하위 리소스가 되는 파드 등을 삭제하기 위해 가비지 수집(Garbage Collection)을 수행
  - ✓ 레플리카셋 등에서는 파드를 생성하지만 생성된 파드에는 어떤 레플리카셋으로 생성되었는지를 판별하기 위해 metadata.ownerReferences 아래에 자동으로 정보를 저장하는데 이것은 디플로이먼트와 레플리카셋에서도 마찬가지
  - ✓ 레플리카셋을 삭제했을 때의 동작
    - Background(기본값): 레플리카셋을 즉시 삭제하고 파드는 가비지 수집기가 백그라운드에서 비동기로 삭제
    - Foreground
      - 레플리카셋을 즉시 삭제하지 않고 deletionTimestamp, metadata.finalizers = foregroundDeletion으로 설정
      - 가비지 수집기가 각 파드에서 blockOwnerDeletion = true인 것을 삭제하는데 false인 것은 Foreground 삭제라도 Background로 삭제
      - 모든 삭제가 끝나면 레플리카셋을 삭제
    - Orphan: 레플리카셋 삭제 시 파드 삭제를 하지 않음
  - ✓ kubectl로 선택할 수 있는 옵션은 Background와 Orphan뿐이며 Foreground를 사용하려면 API를 조작해야 함

리소스 삭제

- 리소스 삭제
  - ✓ 리소스 매니페스트: sample-rs.yaml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: sample-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.17
```

리소스 삭제

- 리소스 삭제
  - ✓ 리소스 배포: kubectl apply -f sample-rs.yaml
  - ✓ 파드 확인: kubectl get pods

```
NAME              READY   STATUS    RESTARTS   AGE
sample-rs-2fvxm   1/1     Running   0          23s
sample-rs-2hkbg   1/1     Running   0          23s
sample-rs-sxtrr   1/1     Running   0          23s
```

  - ✓ 상위 리소스 정보 확인: kubectl get pods sample-rs-sxtrr -o json | jq .metadata.ownerReferences

```
[
  {
    "apiVersion": "apps/v1",
    "blockOwnerDeletion": true,
    "controller": true,
    "kind": "ReplicaSet",
    "name": "sample-rs",
    "uid": "ce0f1715-8689-4353-8069-4ae5f539be54"
  }
]
```

리소스 삭제

- 리소스 삭제
  - ✓ 리소스 연쇄 삭제: kubectl delete --cascade=background replicaset sample-rs
  - ✓ Orphan 삭제
    - 리소스 배포: kubectl apply -f sample-rs.yaml
    - 리소스 삭제: kubectl delete --cascade=orphan replicaset sample-rs
    - 레플리카셋 확인: kubectl get replicasets

      No resources found in default namespace.
    - 파드 확인: kubectl get pods

      NAME READY STATUS RESTARTS AGE

      sample-rs-gjbg6 1/1 Running 0 3m32s

      sample-rs-kxlsj 1/1 Running 0 3m32s

      sample-rs-pkrg6 1/1 Running 0 3m32s

리소스 삭제

- ❖ 리소스 삭제
  - ✓ Foreground 삭제
    - 리소스 배포: kubectl apply -f sample-rs.yaml
    - 리소스 삭제: kubectl delete --cascade=foreground replicaset sample-rs
    - 레플리카셋 확인: kubectl get replicasets --watch
    - 파드 확인: kubectl get pods --watch

Update Strategy

- 업데이트 전략 개요
  - ✓ Deployment의 배포 전략은 주로 애플리케이션이 변경될 때 사용
  - ✓ 이전 버전의 애플리케이션에서 업데이트가 필요한 경우에 주로 사용되며 방법으로는 RollingUpdate, Recreate가 있음
  - ✓ 기본 전략은 RollingUpdate

Update Strategy

- recreate
  - ✓ 재생성(recreate) 업데이트는 모든 이전 버전(V1)의 Pod를 모두 한 번에 종료하고 새 버전(V2)의 Pod로 일괄적으로 교체하는 방식
  - ✓ 빠르게 교체할 수 있지만 새로운 버전의 Pod에 문제가 발생하면 대처가 늦어질 수 있다는 단점이 있음
  - ✓ 재생성 업데이트 설정

```
spec:
  replicas: 3
  strategy:
    type: Recreate
  template:
```

Update Strategy

- ❖ recreate
  - ✓ 실습
    - ● 매니페스트: sample-deployment-recreate.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment-recreate
spec:
  strategy:
    type: Recreate
  replicas: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

Update Strategy

- ❖ recreate
  - ✓ 실습
    - 리소스 생성: kubectl apply -f sample-deployment-recreate.yaml
    - 이미지 변경: kubectl set image deployment sample-deployment-recreate nginx-container=nginx:1.17

      deployment.apps/sample-deployment-recreate image updated

Update Strategy

- recreate
  - ✓ 실습
    - 파드 확인: kubectl get pods --watch

```
NAME                                          READY   STATUS    RESTARTS   AGE
sample-deployment-recreate-556b965888-rk5fq   0/1     Pending   0          0s
sample-deployment-recreate-556b965888-sm5b8   0/1     Pending   0          0s
sample-deployment-recreate-556b965888-rk5fq   0/1     Pending   0          0s
sample-deployment-recreate-556b965888-p6vtr   0/1     ContainerCreating   0
    0s
sample-deployment-recreate-556b965888-sm5b8   0/1     ContainerCreating   0
    0s
sample-deployment-recreate-556b965888-rk5fq   0/1     ContainerCreating   0
    0s
sample-deployment-recreate-556b965888-p6vtr   1/1     Running             0
    2s
sample-deployment-recreate-556b965888-rk5fq   1/1     Running             0
    3s
sample-deployment-recreate-556b965888-sm5b8   1/1     Running             0
    7s
```

Update Strategy

- recreate
  - ✓ 실습
    - 파드 확인: kubectl get pods --watch

```
sample-deployment-recreate-556b965888-p6vtr   1/1     Terminating         0
    26s
sample-deployment-recreate-556b965888-rk5fq   1/1     Terminating         0
    26s
sample-deployment-recreate-556b965888-sm5b8   1/1     Terminating         0
    26s
0s
sample-deployment-recreate-7f9df5d7bf-rhv4f   0/1     Pending             0
    0s
sample-deployment-recreate-7f9df5d7bf-rhv4f   0/1     Pending             0
    0s
sample-deployment-recreate-7f9df5d7bf-zxnhr   0/1     Pending             0
    0s
sample-deployment-recreate-7f9df5d7bf-tvmf8   0/1     ContainerCreating   0
    0s
sample-deployment-recreate-7f9df5d7bf-rhv4f   0/1     ContainerCreating   0
    0s
sample-deployment-recreate-7f9df5d7bf-zxnhr   0/1     ContainerCreating   0
    0s
```

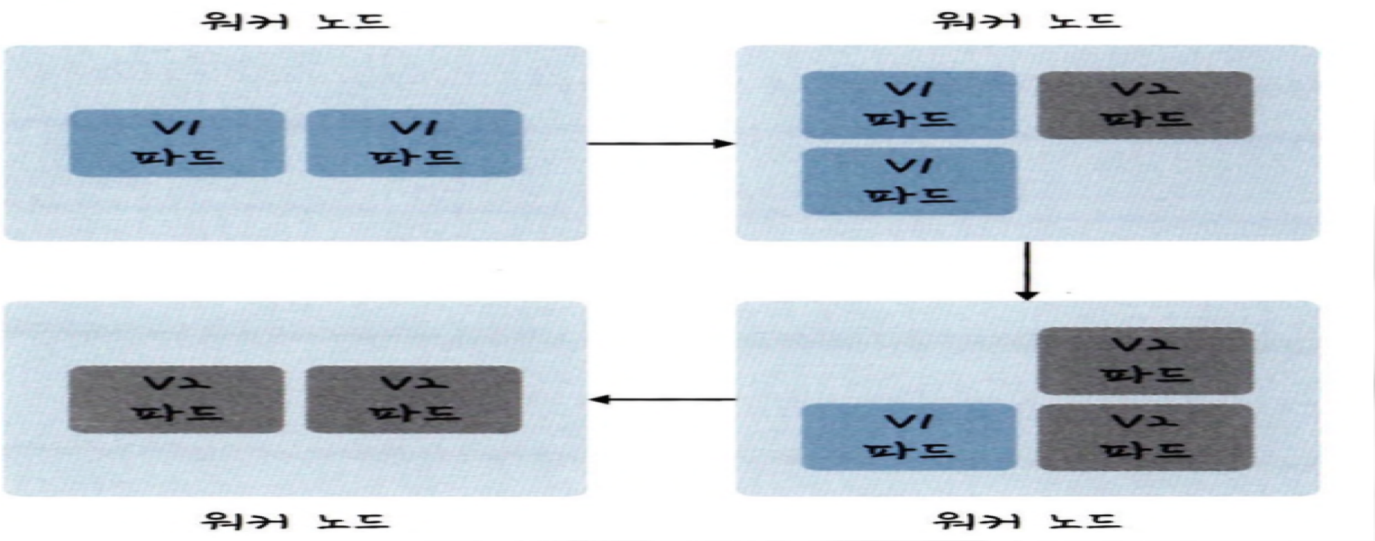
Update Strategy

- recreate
  - ✓ 실습
    - 레플리카셋 확인: kubectl get replicasets --watch

```
NAME                                DESIRED   CURRENT   READY   AGE
sample-deployment-recreate-58754856f   3         3         0       1s
sample-deployment-recreate-79598877b   0         0         0       53s
sample-deployment-recreate-b6cfdc59f   0         0         0       71s
sample-deployment-recreate-58754856f   3         3         2       3s
sample-deployment-recreate-58754856f   3         3         3       3s
```

Update Strategy

- ❖ RollingUpdate
  - ✓ 개요
    - Rolling 업데이트는 새 버전의 애플리케이션을 배포할 때 새 버전의 애플리케이션은 하나씩 늘려가고 기존 버전의 애플리케이션은 하나씩 줄여나가는 방식
    - Kubernetes에서 사용하는 표준 배포 방식
    - 새로운 버전(V2)으로 배포된 Pod에 문제가 발생하면 다시 이전 버전(V1)의 Pod로 서비스를 대체할 수 있어서 상당히 안정적인 배포 방식이지만 업데이트가 느리게 진행된다는 단점이 있음



Update Strategy

- RollingUpdate
  - ✓ 설정 값
    - maxSurge
      - Rolling 업데이트를 위해 최대로 생성할 수 있는 Pod 갯수
      - maxSurge를 높게 설정하면 Rolling 배포를 빠르게 적용 가능
      - % 또는 갯수로 지정 가능
      - 설정하지 않을 경우 기본 값은 25%
    - maxUnavailable
      - Rolling 업데이트 중 최대로 삭제할 Pod 갯수
      - maxUnavailable를 높게 설정하면 Rolling 배포를 빠르게 적용 가능
      - 한번에 많은 Pod를 삭제할 경우 트래픽이 남아있는 소수의 Pod로 집중될 수 있어 값을 적절히 설정 필요
      - % 또는 갯수로 지정 가능
      - 설정하지 않을 경우 기본 값은 25%

Update Strategy

- RollingUpdate
  - ✓ 실습
    - 매니페스트: sample-deployment-rollingupdate.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment-rollingupdate
```

Update Strategy

- RollingUpdate
  - ✓ 실습
    - 매니페스트: sample-deployment-rollingupdate.yaml

```
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
  replicas: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

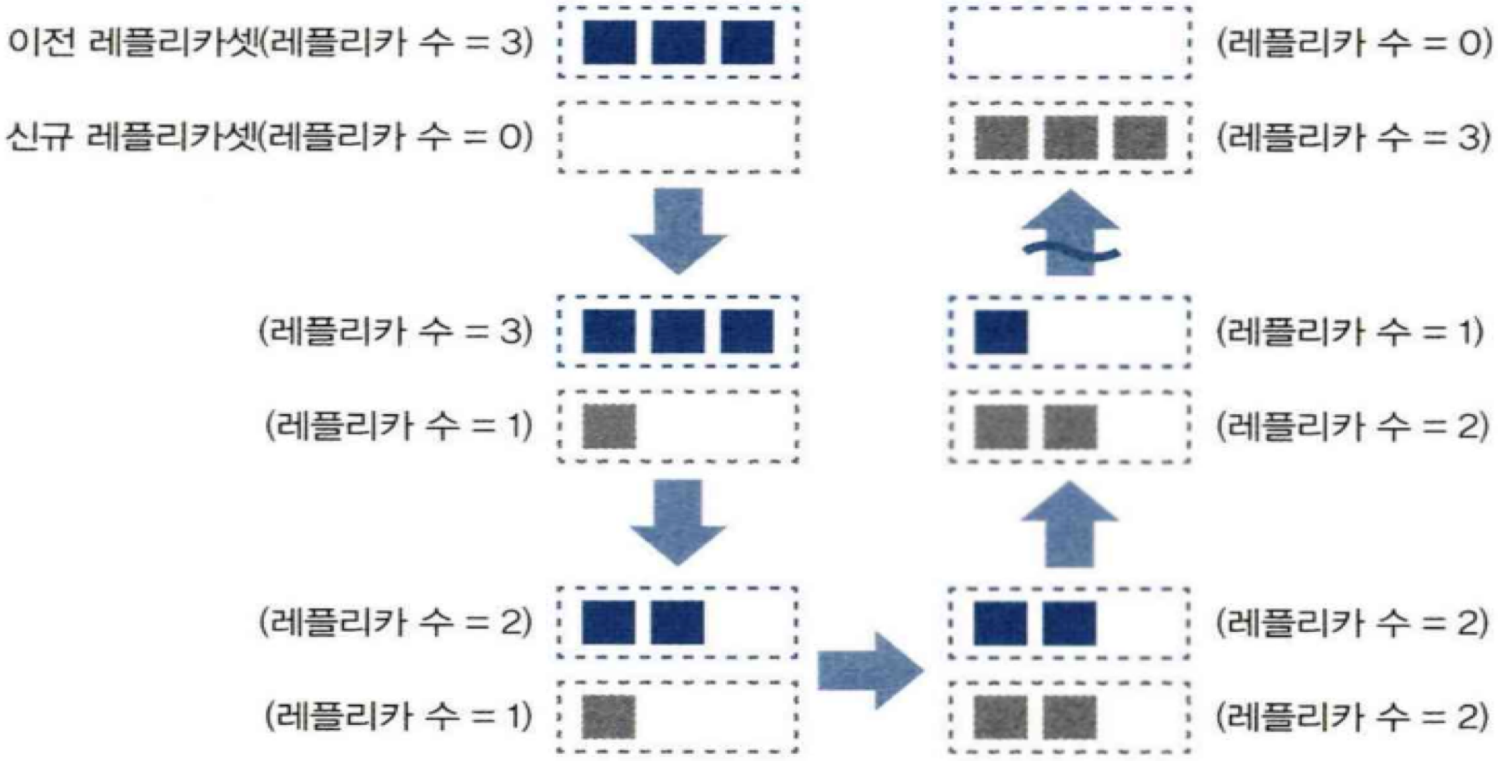
Update Strategy

- RollingUpdate
  - ✓ 실습
    - 리소스 배포: kubectl apply -f sample-deployment-rollingupdate.yaml
    - 이미지 변경: kubectl set image deployment sample-deployment-rollingupdate nginx-container=nginx:1.17
    - 변경 내역 확인: kubectl get replicasets --watch

```
NAME                                          DESIRED   CURRENT   READY   AGE
sample-deployment-rollingupdate-79598877b     3         3         2       7s
sample-deployment-rollingupdate-b6cfdc59f     1         1         1       20s
sample-deployment-rollingupdate-79598877b     3         3         3       9s
sample-deployment-rollingupdate-b6cfdc59f     0         1         1       22s
sample-deployment-rollingupdate-b6cfdc59f     0         1         1       22s
sample-deployment-rollingupdate-b6cfdc59f     0         0         0       22s
```

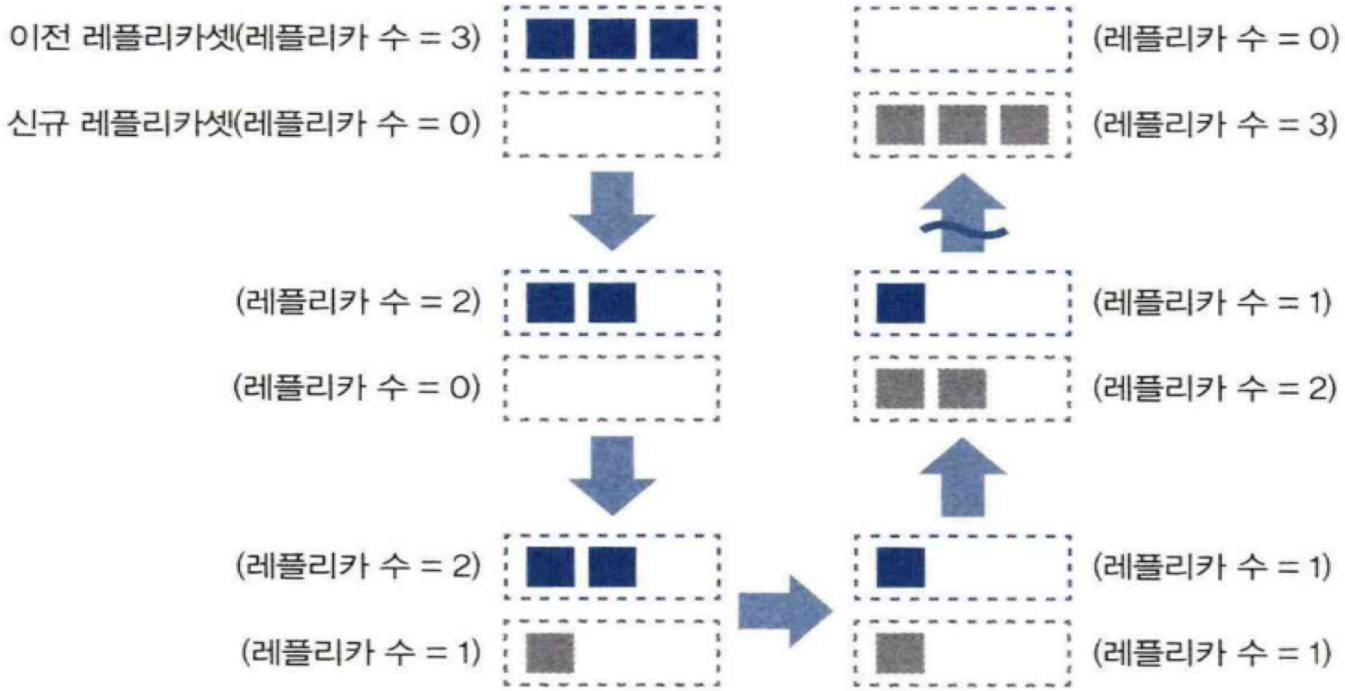
Update Strategy

- ❖ RollingUpdate
  - ✓ 설정 값
    - maxUnavailable=0/maxSurge=1의 롤링 업데이트



Update Strategy

- RollingUpdate
  - ✓ 설정 값
    - maxUnavailable=1/maxSurge=0의 롤링 업데이트



Update Strategy

- RollingUpdate
  - ✓ DaemonSet 업데이트 전략: OnDelete & RollingUpdate
    - OnDelete
      - OnDelete에서는 DaemonSet 매니페스트를 수정하여 이미지 등을 변경했더라도 기존 파드는 업데이트되지 않음
      - Deployment와 달리 DaemonSet은 모니터링이나 로그 전송과 같은 용도로 많이 사용하기 때문에 업데이트는 다음 번에 다시 생성할 때나 수동으로 임의의 시점에 하게 되어 있으며 type 외에 지정할 수 있는 항목은 없음
      - OnDelete로 설정하고 아무것도 하지 않을 경우 노드의 유지 보수를 이유로 파드가 정지되거나 예기치 못하게 파드가 정지되는 등 다양한 이유로 파드가 정지되어 다시 생성되기 전까지는 업데이트 되지 않기 때문에 운영상의 이유로 정지하면 안 되는 파드이거나 업데이트가 급하게 필요 없는 경우 OnDelete 설정으로 사용해도 되지만 이전 버전이 계속 장기간 사용된다는 점에 주의
      - 임의의 시점에 파드를 업데이트하는 경우에는 DaemonSet과 연결된 파드를 kubectl delete pod 명령어로 수동으로 정지시키고 자동화된 복구 기능으로 새로운 파드를 생성해야 함

Update Strategy

- RollingUpdate
  - ✓ DaemonSet 업데이트 전략: OnDelete & RollingUpdate
    - OnDelete
      - 매니페스트: sample-ds-ondelete.yaml

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: sample-ds-ondelete
spec:
  updateStrategy:
    type: OnDelete
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

Update Strategy

- RollingUpdate
  - ✓ DaemonSet 업데이트 전략: OnDelete & RollingUpdate
    - OnDelete
      - ◆ 리소스 생성

        kubectl apply -f sample-ds-ondelete.yaml

        daemonset.apps/sample-ds-ondelete created
      - ◆ 리소스 확인

        kubectl get pods

        ```
        NAME                       READY   STATUS    RESTARTS   AGE
        sample-ds-ondelete-2kbgx   1/1     Running   0          5s
        sample-ds-ondelete-8bnrw   1/1     Running   0          5s
        sample-ds-ondelete-r8mqt   1/1     Running   0          5s
        ```

Update Strategy

- RollingUpdate
  - ✓ DaemonSet 업데이트 전략: OnDelete & RollingUpdate
    - RollingUpdate
      - DaemonSet도 Deployment와 마찬가지로 RollingUpdate를 사용한 업데이트가 가능
      - DaemonSet에서는 하나의 노드에 동일한 파드를 여러 개 생성할 수 없으므로 Deployment와 달리 동시에 생성할 수 있는 최대 파드 수(maxSurge)를 설정할 수 없고 동시에 정지 가능한 최대 파드수(maxUnavailable)만 지정하여 RollingUpdate를 수행
      - maxUnavailable=2의 경우 파드를 두 개씩 동시에 업데이트해 나가는 형태
      - maxUnavailable의 기본값은 1이며 maxUnavailable을 0으로 지정할 수는 없음

Update Strategy

- ❖ RollingUpdate
  - ✓ DaemonSet 업데이트 전략: OnDelete & RollingUpdate
    - ● RollingUpdate
      - ◆ 매니페스트: sample-ds-rollingupdate.yaml

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: sample-ds-rollingupdate
spec:
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 2
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

Update Strategy

- RollingUpdate
  - ✓ StatefulSet 업데이트 전략
    - OnDelete
      - OnDelete에서는 스테이트풀셋 매니페스트를 수정하여 이미지 등을 변경했더라도 기존 파드는 업데이트되지 않음
      - 스테이트풀셋에서 OnDelete는 영속성 영역을 가진 데이터베이스나 클러스터 등에서 많이 사용되기 때문에 수동으로 업데이트하고 싶을 경우 OnDelete를 사용하여 임의의 시점에서나 다음에 재기동할 때 업데이트를 진행하게 되어 있는데 type 외에 지정할 수 있는 항목은 없음

Update Strategy

- RollingUpdate
  - ✓ StatefulSet 업데이트 전략
    - OnDelete
      - ◆ 매니페스트: sample-statefulset-ondelete.yaml

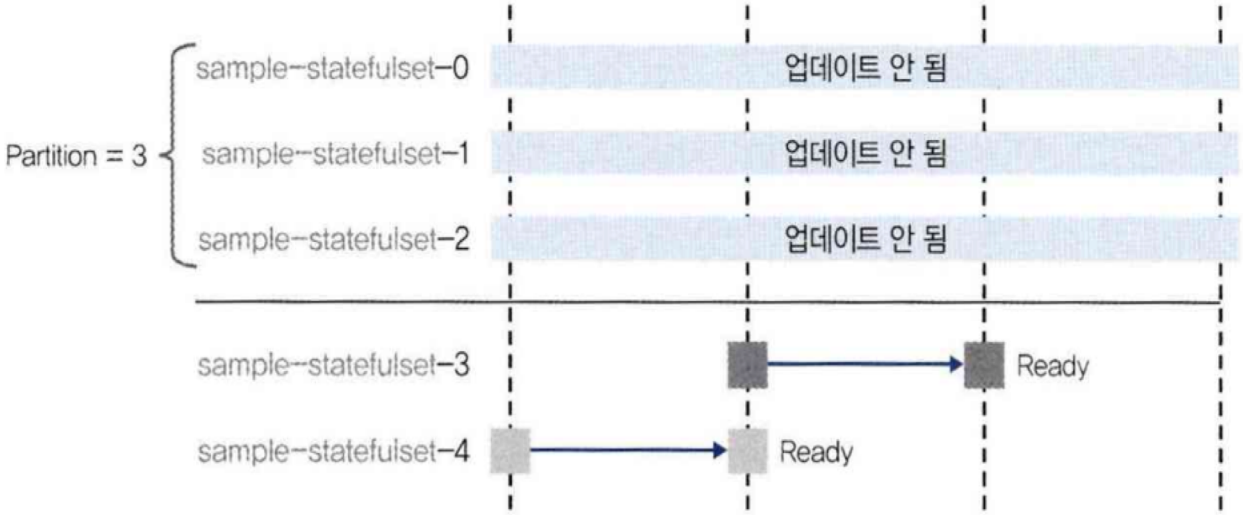
```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sample-statefulset-ondelete
spec:
  updateStrategy:
    type: OnDelete
  serviceName: sample-statefulset-ondelete
  replicas: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

Update Strategy

- ❖ RollingUpdate
  - ✓ StatefulSet 업데이트 전략
    - RollingUpdate
      - ◆ Deployment와 마찬가지로 RollingUpdate를 사용한 스테이트풀셋 업데이트가 가능하지만 스테이트풀셋에는 영속성 데이터가 있으므로 Deployment와 다르게 추가 파드를 생성해서 롤링 업데이트를 할 수 없고 동시에 정지 가능한 최대 파드 수(maxUnavailable)를 지정하여 롤링 업데이트를 할 수도 없으므로 파드마다 Ready 상태인지를 확인하고 업데이트
      - ◆ spec.podManagementPolicy가 Parallel로 설정되어 있는 경우에도 병렬로 동시에 처리되지 않고 파드 하나씩 업데이트
      - ◆ 스테이트풀셋의 RollingUpdate에서는 partition이라는 특정 값을 설정할 수 있는데 partition을 설정하면 전체 파드 중 어떤 파드까지 업데이트할지를 지정할 수 있고 이 설정을 사용하면 전체에 영향을 주지 않고 부분적으로 업데이트를 적용하고 확인할 수 있어 안전한 업데이트를 할 수 있음
      - ◆ OnDelete와 달리 수동으로 재기동한 경우에도 partition 값보다 작은 인덱스를 가진 파드는 업데이트되지 않음

Update Strategy

- ❖ RollingUpdate
  - ✓ StatefulSet 업데이트 전략
    - ● RollingUpdate



Update Strategy

- ❖ RollingUpdate
  - ✓ StatefulSet 업데이트 전략
    - ● RollingUpdate
      - ◆ 매니페스트: sample-statefulset-rollingupdate.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sample-statefulset-rollingupdate
spec:
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 3
  serviceName: sample-statefulset-rollingupdate
  replicas: 5
  selector:
    matchLabels:
      app: sample-app
```

Update Strategy

- ❖ RollingUpdate
  - ✓ StatefulSet 업데이트 전략
    - ● RollingUpdate
      - ◆ 매니페스트: sample-statefulset-rollingupdate.yaml

```
template:
  metadata:
    labels:
      app: sample-app
  spec:
    containers:
    - name: nginx-container
      image: nginx
```

Update Strategy

- ❖ RollingUpdate
  - ✓ StatefulSet 업데이트 전략
    - ● RollingUpdate
      - ◆ 배포: kubectl apply –f sample-statefulset-rollingupdate.yaml
      - ◆ 파드 확인: kubectl get pods

```
NAME                                READY   STATUS    RESTARTS   AGE
sample-statefulset-rollingupdate-0  1/1     Running   0          2m49s
sample-statefulset-rollingupdate-1  1/1     Running   0          2m45s
sample-statefulset-rollingupdate-2  1/1     Running   0          2m42s
sample-statefulset-rollingupdate-3  1/1     Running   0          2m39s
sample-statefulset-rollingupdate-4  1/1     Running   0          2m36s
```

Update Strategy

- RollingUpdate
  - ✓ StatefulSet 업데이트 전략
    - RollingUpdate
      - 이미지 수정: kubectl set image statefulset/sample-statefulset-rollingupdate nginx-container=nginx:1.17
      - 파드 수정 확인: kubectl get pods --watch

```
NAME                                READY   STATUS              RESTARTS   AGE
sample-statefulset-rollingupdate-0  1/1     Running             0          3m39s
sample-statefulset-rollingupdate-1  1/1     Running             0          3m35s
sample-statefulset-rollingupdate-2  1/1     Running             0          3m32s
sample-statefulset-rollingupdate-3  0/1     ContainerCreating   0          2s
sample-statefulset-rollingupdate-4  1/1     Running             0          7s
sample-statefulset-rollingupdate-3  1/1     Running             0          3s
```

Update Strategy

- RollingUpdate
  - ✓ 안전하게 애플리케이션을 중지하고 파드 삭제
    - 쿠버네티스의 RollingUpdate는 차례로 파드를 중지하고 새로운 파드로 교체하는 동작을 수행하는데 이는 강력한 구조이지만 파드를 삭제할 때는 주의해야 할 점이 존재하는데 파드의 컨테이너가 삭제될 때 유저의 HTTP Request를 처리 중이라면 유저는 기대한 응답을 받을 수 없음
    - 데이터 스토어라면 종료까지 어느 정도 시간이 걸릴 때도 있으므로 종료 처리의 완료를 기다리지 않고 파드가 삭제할 수 있음
    - 애플리케이션을 안전하게 중지하고 파드 내부의 컨테이너가 삭제되도록 제어하는 것이 중요한데 애플리케이션을 정상적인 방법을 거쳐 안전하게 종료하는 것을 Graceful Shutdown 이라고 함
    - 파드에 종료 명령이 전달되면 파드에 속하는 컨테이너 프로세스에는 SIGTERM이 전송되는데 SIGTERM을 받은 컨테이너는 terminationGracePeriodSeconds로 설정된(기본값은 30초) 이내에 애플리케이션이 정상 종료되지 않으면 SIGKILL이 전송되고 컨테이너는 강제로 종료됨
    - 종료 처리에 시간이 걸리는 것 과 같은 컨테이너는 terminationGracePeriodSeconds의 값을 길게 설정하는 것이 좋음 - terminationGracePeriodSeconds를 60초 정도로 설정

Update Strategy

- RollingUpdate
  - ✓ 안전하게 애플리케이션을 중지하고 파드 삭제
    - nginx는 다른 고려사항이 필요한데 nginx는 SIGTERM을 받으면 Graceful Shutdown이 아니라 즉시 종료하는데 게다가 master 프로세스 뿐 만 아니라 worker 프로세스도 동작하고 있으므로 모두 안전하게 종료해야 함
    - 쿠버네티스는 lifecycle.preStop으로 컨테이너가 종료되기 전에 훅 작업을 정의할 수 있는데 이를 사용해 nginx를 안전하게 종료하는 커맨드인 quit를 실행

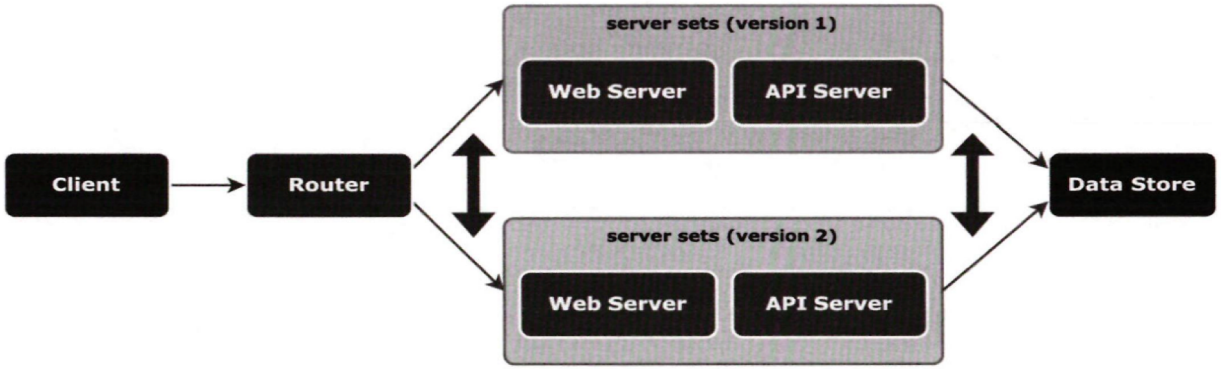
```
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    lifecycle:
      preStop:
        exec:
          command:
            - "/bin/sh"
            - "-c"
            - "nginx -s quit"
```

Update Strategy

- ❖ 블루/그린 업데이트
  - ✓ 개요
    - RollingUpdate 구조는 강력하지만 신버전 과 구버전의 파드가 혼재하는 시간이 발생하는데 이 특성은 애플리케이션의 사용자나 다른 프로그램에 의해 의도하지 않은 부작용이 발생할 수 있으므로 RollingUpdate가 적절하지 않은 애플리케이션도 있음
    - 블루/그린(blue/green) 업데이트는 애플리케이션의 이전 버전(블루, V1 Pod)과 새 버전(그린, V2 Pod)을 동시에 운영
    - 서비스 목적으로 접속할 수 있는 것은 새 버전의 Pod만 가능하며 이전 버전의 Pod는 테스트 목적으로만 접속할 수 있음
    - 새로운 버전의 Pod에 문제가 발생했을 때 빠르게 대응할 수 있으며 안정적으로 배포할 수 있음
    - 도커와 쿠버네티스는 서버 그룹이 아니라 컨테이너 그룹이지만 일시적이라도 동시에 두 그룹의 서버를 사용하게 되므로 필요한 리소스는 RollingUpdate 보다 많아지지만 두 버전이 혼재하는 시간 없이 바로 교체할 수 있으며 한쪽의 서버 그룹을 배포 전 Standby 상태로 이용할 수 있으므로 운영 시에도 장점이 큰 배포 방법

Update Strategy

- 블루/그린 업데이트
  - ✓ 개요
    - Blue-Green Deploymentsms 배포시 기존에 배포된 서버 그룹과는 별도로 새로운 애플리케이션을 배포할 서버 그룹을 준비하고 로드 밸런서와 서비스 디스커버리 레벨에서 참조 대상을 교체하여 배포하는 방법
    - 블루/그린 업데이트를 이용해 애플리케이션을 배포할 때는 version 으로 구분해 관리
    - green 배포용 yaml 파일을 생성하는데 이전 버전의 배포 yaml 파일과 내용이 같지만 사용하는 이미지의 경로와 labels의 version만 수정



Update Strategy

- ❖ 블루/그린 업데이트
  - ✓ 실습
    - Blue 버전의 매니페스트: print-version-blue.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: print-version-blue
  labels:
    app: print-version
    color: blue
spec:
  replicas: 1
  selector:
    matchLabels:
      app: print-version
      color: blue
```

Update Strategy

- ❖ 블루/그린 업데이트
  - ✓ 실습
    - Blue 버전의 매니페스트: print-version-blue.yaml

```
template:
  metadata:
    labels:
      app: print-version
      color: blue
  spec:
    containers:
    - name: print-version
      image: ghcr.io/jpubdocker/print-version:v0.0.1
      ports:
      - containerPort: 8080
```

Update Strategy

- ❖ 블루/그린 업데이트
  - ✓ 실습
    - Green 버전의 매니페스트: print-version-green.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: print-version-green
  labels:
    app: print-version
    color: green
spec:
  replicas: 1
  selector:
    matchLabels:
      app: print-version
      color: green
```

Update Strategy

- ❖ 블루/그린 업데이트
  - ✓ 실습
    - Green 버전의 매니페스트: print-version-green.yaml

```
template:
  metadata:
    labels:
      app: print-version
      color: green
  spec:
    containers:
    - name: print-version
      image: ghcr.io/jpubdocker/print-version:v0.0.2
      ports:
      - containerPort: 8080
```

Update Strategy

- 블루/그린 업데이트
  - ✓ 실습
    - 리소스 생성
      - kubectl apply -f print-version-blue.yaml
      - kubectl apply -f print-version-green.yaml

Update Strategy

- ❖ 블루/그린 업데이트
  - ✓ 실습
    - ● 서비스 생성
      - ◆ 서비스 매니페스트: print-version-service-coloy.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: print-version
  labels:
    app: print-version
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: print-version
    color: blue
```

        - ▪ 서비스는 spec.label로 레이블을 지정하여 Request를 전송할 디플로이먼트를 결정할 수 있도록 작성
      - ◆ 서비스 리소스 배포: kubectl apply -f print-version-service-coloy.yaml

Update Strategy

- ❖ 블루/그린 업데이트
  - ✓ 실습
    - ● 업데이트 확인
      - ◆ print-version 서비스에 GET 요청을 전송하는 매니페스트: update-checker.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: update-checker
  labels:
    app: update-checker
spec:
  containers:
  - name: kubectl
    image: ghcr.io/jpubdocker/debug:v0.1.0
    command:
    - sh
    - -c
    - |
      while true
      do
        VERSION=`curl -s http://print-version/`
        echo "[`date`] $VERSION"
        sleep 1
      done
```

Update Strategy

- ❖ 블루/그린 업데이트
  - ✓ 실습
    - ● 업데이트 확인
      - ◆ print-version 서비스에 GET 요청을 전송하는 리소스 배포: kubectl apply -f update-checker.yaml
      - ◆ 로그 확인: kubectl logs -f update-checker

        [Tue Feb 24 23:37:07 UTC 2026] VERSION=v0.0.1

        [Tue Feb 24 23:37:08 UTC 2026] VERSION=v0.0.1

        [Tue Feb 24 23:37:09 UTC 2026] VERSION=v0.0.1

        [Tue Feb 24 23:37:10 UTC 2026] VERSION=v0.0.1

        [Tue Feb 24 23:37:11 UTC 2026] VERSION=v0.0.1

        [Tue Feb 24 23:37:12 UTC 2026] VERSION=v0.0.1

        [Tue Feb 24 23:37:13 UTC 2026] VERSION=v0.0.1

        [Tue Feb 24 23:37:14 UTC 2026] VERSION=v0.0.1

Update Strategy

- ❖ 블루/그린 업데이트
  - ✓ 실습
    - ● 업데이트 확인
      - ◆ 버전 변경을 위한 서비스 수정: kubectl edit service/print-version

```
spec:
  clusterIP: 10.97.244.230
  clusterIPs:
  - 10.97.244.230
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: print-version
    color: green
```

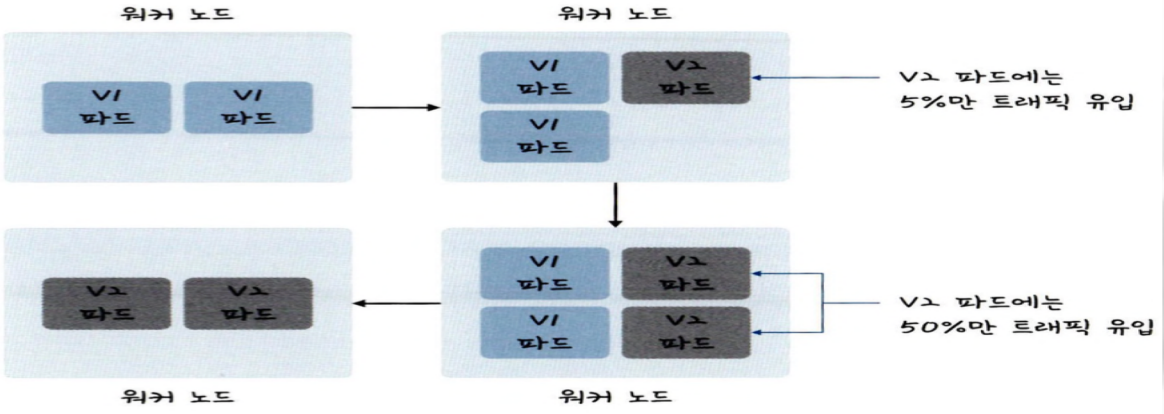
Update Strategy

- ❖ 블루/그린 업데이트
  - ✓ 실습
    - ● 업데이트 확인
      - ◆ 로그 확인: kubectl logs -f update-checker

[Tue Feb 24 23:37:07 UTC 2026] VERSION=v0.0.2
[Tue Feb 24 23:37:08 UTC 2026] VERSION=v0.0.2
[Tue Feb 24 23:37:09 UTC 2026] VERSION=v0.0.2
[Tue Feb 24 23:37:10 UTC 2026] VERSION=v0.0.2
[Tue Feb 24 23:37:11 UTC 2026] VERSION=v0.0.2
[Tue Feb 24 23:37:12 UTC 2026] VERSION=v0.0.2
[Tue Feb 24 23:37:13 UTC 2026] VERSION=v0.0.2
[Tue Feb 24 23:37:14 UTC 2026] VERSION=v0.0.2

Update Strategy

- Canary 업데이트
  - ✓ 개요
    - Canary 업데이트는 블루/그린과 비슷하지만 더 진보적인 단계적 접근 방식
    - Canary는 주로 애플리케이션의 몇몇 새로운 기능을 테스트할 때 사용
    - Canary는 두 개의 버전을 모두 배포하지만 새 버전에는 조금씩 트래픽을 증가시켜(처음에는 5% 정도의 트래픽만 흘려 보내지만 나중에는 50% 의 트래픽을 흘려보냄) 새로운 기능들을 테스트
    - 기능 테스트가 끝나고 새 버전에 문제가 없다고 판단하면 이전 버전은 모두 종료시키고 새 버전으로만 서비스



Update Strategy

- ❖ Canary 업데이트
  - ✓ Canary 업데이트의 문제점
    - UI/UX가 변경되어 100명의 유저가 사용하는 서비스에서 20명을 대상으로 Canary 업데이트를 진행하고 싶은 경우 가장 일반적인 방식 중 하나는 로드 밸런서를 통해서 서비스의 엔드포인트를 유저들에게 노출하는 것인데 로드 밸런서는 서버의 과부하가 일어나지 않는 가장 최적화된 방식으로 트레픽을 분산하는데 한 명의 유저가 서비스를 사용할 때 로드 밸런서에 의해 자동으로 여러 서버에 접속될 수 있음
    - 어떤 유저가 웹페이지를 새로고침 했을 때 로드 밸런싱이 일어나 Canary 업데이트가 있는 서버에서 기존 서버로 바뀌었다고 가정하면 갑자기 신버전 UI/UX에서 구버전으로 변경되게 되는데 서비스를 사용하기 불편할 뿐만 아니라 UI/UX에서 사용자 반응성이나 이용 시간 등 일련의 과정을 거쳐야 하는 테스트를 진행하기 힘들 수 있음
    - 스티키 세션을 사용하여 사용자가 한번 특정한 서버에 접속되어 있으면 그 서버에만 계속 접속되도록 만들 수 있음

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● 기본 Service와 Pod 비율을 이용한 방식을 이용한 구현 – 가장 단순
      - ◆ stable 버전 매니페스트: canary-stable.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-v1
  labels:
    app: print-version
    version: v1
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● 기본 Service와 Pod 비율을 이용한 방식을 이용한 구현 – 가장 단순
      - ◆ stable 버전 매니페스트: canary-stable.yaml

```
spec:
  replicas: 3
  selector:
    matchLabels:
      app: print-version
      version: v1
  template:
    metadata:
      labels:
        app: print-version
        version: v1
    spec:
      containers:
      - name: print-version
        image: ghcr.io/jpubdocker/print-version:v0.0.1
        ports:
        - containerPort: 8080
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● 기본 Service와 Pod 비율을 이용한 방식을 이용한 구현 – 가장 단순
      - ◆ stable 버전 매니페스트: canary-new.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-canary
  labels:
    app: print-version
    version: v2
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● 기본 Service와 Pod 비율을 이용한 방식을 이용한 구현 – 가장 단순
      - ◆ stable 버전 매니페스트: canary-new.yaml

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: print-version
      version: v2
  template:
    metadata:
      labels:
        app: print-version
        version: v2
    spec:
      containers:
      - name: print-version
        image: ghcr.io/jpubdocker/print-version:v0.0.2
        ports:
        - containerPort: 8080
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● 기본 Service와 Pod 비율을 이용한 방식을 이용한 구현 – 가장 단순
      - ◆ 서비스 매니페스트: canary-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: print-version
  labels:
    app: print-version
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: print-version
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● 기본 Service와 Pod 비율을 이용한 방식을 이용한 구현 – 가장 단순
      - ◆ 리소스 배포
        - kubectl apply -f canary-stable.yaml
        - kubectl apply -f canary-new.yaml
        - kubectl apply -f canary-service.yaml
        - kubectl apply -f update-checker.yaml
      - ◆ 로그 확인: kubectl logs -f update-checker

```
[Wed Feb 25 05:12:57 UTC 2026] VERSION=cb76260
[Wed Feb 25 05:12:58 UTC 2026] VERSION=cb76260
[Wed Feb 25 05:12:59 UTC 2026] VERSION=cb76260
[Wed Feb 25 05:13:00 UTC 2026] VERSION=cb76260
[Wed Feb 25 05:13:01 UTC 2026] VERSION=cb76260
[Wed Feb 25 05:13:02 UTC 2026] VERSION=cb76260
[Wed Feb 25 05:13:03 UTC 2026] VERSION=v0.0.2
[Wed Feb 25 05:13:04 UTC 2026] VERSION=cb76260
[Wed Feb 25 05:13:05 UTC 2026] VERSION=cb76260
[Wed Feb 25 05:13:06 UTC 2026] VERSION=cb76260
[Wed Feb 25 05:13:07 UTC 2026] VERSION=cb76260
[Wed Feb 25 05:13:08 UTC 2026] VERSION=v0.0.2
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● Nginx Ingress Controller의 Annotations 기능을 이용한 구현 – 권장
      - ◆ 기존 버전 매니페스트: main.yaml

```
# 1-1. Main 버전 (v1)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-v1
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
      version: v1
  template:
    metadata:
      labels:
        app: my-app
        version: v1
    spec:
      containers:
      - name: app
        image: ghcr.io/jpubdocker/print-version:v0.0.1
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● Nginx Ingress Controller의 Annotations 기능을 이용한 구현 – 권장
      - ◆ 기존 버전 매니페스트: main.yaml

```
---
# 1-2. Main 서비스
apiVersion: v1
kind: Service
metadata:
  name: app-main-svc
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: my-app
    version: v1
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● Nginx Ingress Controller의 Annotations 기능을 이용한 구현 – 권장
      - ◆ Canary 버전 매니페스트: canary.yaml

```
# 2-1. Canary 버전 (v2)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-canary
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
      version: v2
  template:
    metadata:
      labels:
        app: my-app
        version: v2
    spec:
      containers:
      - name: app
        image: ghcr.io/jpubdocker/print-version:v0.0.2
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● Nginx Ingress Controller의 Annotations 기능을 이용한 구현 – 권장
      - ◆ Canary 버전 매니페스트: canary.yaml

```
---
# 2-2. Canary 서비스
apiVersion: v1
kind: Service
metadata:
  name: app-canary-svc
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: my-app
    version: v2
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● Nginx Ingress Controller의 Annotations 기능을 이용한 구현 – 권장
      - ◆ Main 버전 인그레스 매니페스트: main-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-main-ingress
spec:
  ingressClassName: nginx
  rules:
  - host: app.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: app-main-svc
            port:
              number: 80
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● Nginx Ingress Controller의 Annotations 기능을 이용한 구현 – 권장
      - ◆ Carnary 버전 인그레스 매니페스트: canary-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-canary-ingress
  annotations:
    # Canary 설정 활성화
    nginx.ingress.kubernetes.io/canary: "true"

    # 10%의 트래픽만 이 인그레스로 분산 (0~100 사이 정수)
    nginx.ingress.kubernetes.io/canary-weight: "10"
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● Nginx Ingress Controller의 Annotations 기능을 이용한 구현 – 권장
      - ◆ Carnary 버전 인그레스 매니페스트: canary-ingress.yaml

```
spec:
  ingressClassName: nginx
  rules:
  - host: app.example.com # 메인과 동일한 호스트명 사용
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: app-canary-svc # v2 서비스로 연결
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● Argo Rollouts을 이용한 방식 – 전문 도구
      - ◆ 현업에서 가장 많이 쓰이는 방식으로 Deployment 대신 Rollout이라는 커스텀 리소스를 사용해 자동화된 카나리를 구현
      - ◆ 매니페스트

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: my-app-rollout
spec:
  replicas: 10
  strategy:
    canary:
      steps:
      - setWeight: 10    # 10% 배포 후 대기
      - pause: {duration: 1h} # 1시간 동안 모니터링
      - setWeight: 50    # 이상 없으면 50%로 확대
      - pause: {}        # 수동 승인 대기
  template:
    # ... (기존 Deployment의 template과 동일)
```

Update Strategy

- ❖ Canary 업데이트
  - ✓ 업데이트 전략
    - ● 선택
      - ◆ 테스트용/단순함: Service Selector
      - ◆ 운영 환경/가중치 조절 필요: Ingress Annotations
      - ◆ 대규모/자동화된 배포 파이프라인: Argo Rollouts

Update Strategy

- 상세 업데이트 파라미터
  - ✓ minReadySeconds(최소 대기 시간- 초): 파드가 Ready 상태가 된 다음부터 디플로이먼트 리소스에서 파드 기동이 완료되었다고 판단(다음 파드의 교체가 가능하다고 판단)하기까지의 최소 시간
  - ✓ revisionHistoryLimit(수정 버전 기록 제한)
    - 디플로이먼트가 유지할 레플리카셋 수
    - 롤백이 가능한 이력 수
  - ✓ progressDeadlineSeconds(진행 기한 시간(초))
    - Recreate/RollingIlpdate 처리 타임아웃 시간
    - 타임아웃 시간이 경과하면 자동으로 롤백

Update Strategy

- 상세 업데이트 파라미터
  - ✓ 매니페스트: sample-deployment-params.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment-params
spec:
  minReadySeconds: 0
  revisionHistoryLimit: 2
  progressDeadlineSeconds: 3600
  replicas: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
      - name: nginx-container
        image: nginx
```