

Section 13: Kustomize Basics

275. Kustomize Problem Statement & ideology

```
├── base/
│   ├── deployment.yaml
│   ├── service.yaml
│   └── kustomization.yaml
├── overlays/
│   ├── dev/
│   │   └── kustomization.yaml
│   ├── staging/
│   │   └── kustomization.yaml
│   └── production/
│       └── kustomization.yaml
```

- base: 모든 환경에서 동일하게 사용할 리소스나 구성
- overlay: 각 환경별로 동작을 사용자 지정
 - base에서 덮어쓰거나 변경하려는 모든 속성, 매개변수를 지정 가능
- kustomize를 사용하면 helm에서처럼 템플릿 시스템을 사용하지 않고, 템플릿 언어를 배울 필요 없이 표준 yaml 기반의 기본구성과 오버레이만 구성하면 되기 때문에 편리하다

276. Kustomize vs Helm

- Helm
 - template 기반 (Go Template 사용) → 모든 것이 변수인 템플릿 구문은 가독성이 떨어짐
 - 환경별 구성을 사용자정의하게 해줄 뿐만 아니라, package manager 역할을 함
 - conditionals, loops, functions, hooks 등 Kustomize에는 없는 복잡한 추가 기능 제공
- Kustomize
 - 일반적인 yaml 기반이기 때문에 helm 보다 가독성이 좋고 간단하다

trade-off가 있기에 각 장단점을 고려하여 사용하자

277. Installation/Setup

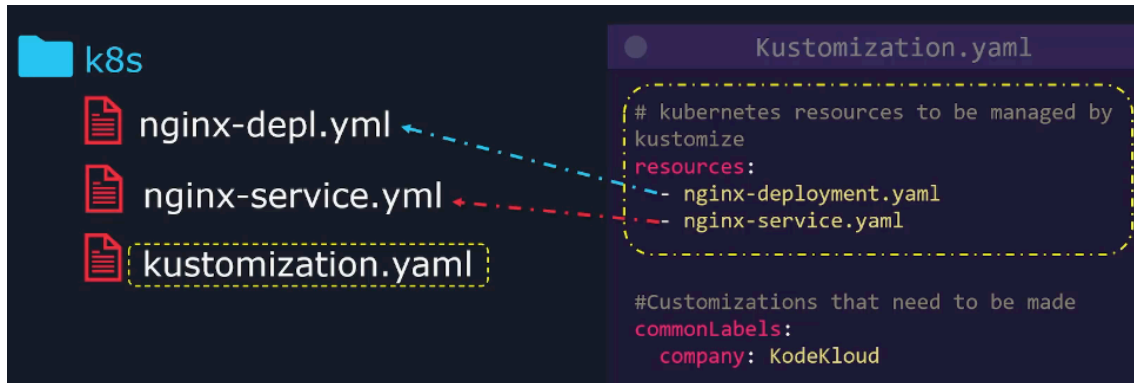
```
curl -s "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install_kustomize.sh" | bash
```

or `brew install kustomize` (Mac)

```
kustomize version --short : 정상 설치 확인
```

278. kustomization.yaml file

- Kustomize를 사용하여 만드는 모든 리소스에 공통 레이블을 추가하는 간단한 예제



- `k8s` 디렉토리 하위에 `kustomization.yaml` 를 직접 생성

```
resources:
- nginx-deployment.yaml
- nginx-service.yaml

commonLabels:
  company: KodeKloud
```

- `kustomize build k8s/` : kustomize 빌드 명령 실행
 - 터미널 output을 보면 공통적인 label을 적용한 것을 확인할 수 있음
 - 하지만, 이 명령을 실행할 때 실제로 쿠버네티스 리소스를 적용하거나 배포하는 것은 아니라는 것을 주의 (터미널에서 최종 구성이 어떻게 될지 보기만 하는 것임) (적용하려면 이 명령의 출력을 가져와서 `kubectl apply` , 다음 장 참고)

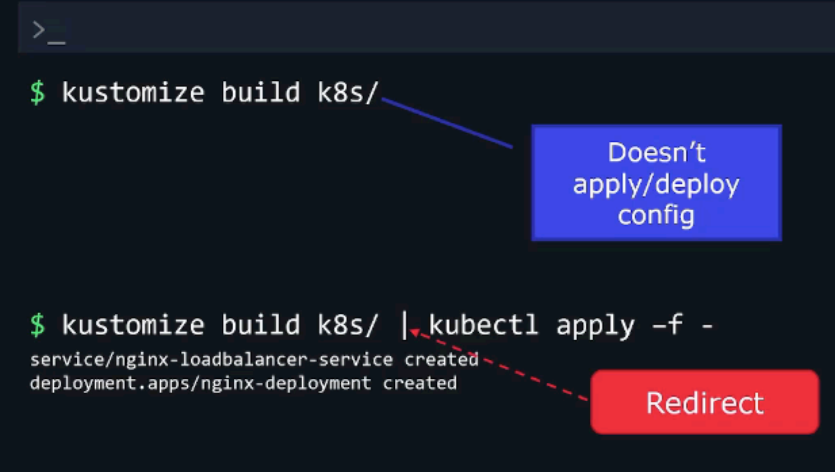
```
>_
$ kustomize build k8s/

kustomize build k8s/
apiVersion: v1
kind: Service
metadata:
  labels:
    company: KodeKloud
  name: nginx-lb-svc
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 3000
  selector:
    company: KodeKloud
    component: nginx
  type: LoadBalancer

---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    company: KodeKloud
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      company: KodeKloud
      component: nginx
  template:
    metadata:
      labels:
        company: KodeKloud
        component: nginx
    spec:
      containers:
        - image: nginx
          name: nginx
```

The terminal output shows the result of running `kustomize build k8s/`. It displays the YAML for a Service and a Deployment. Both resources have a common label `company: KodeKloud` applied. The Service is named `nginx-lb-svc` and the Deployment is named `nginx-deployment`. The Deployment has 1 replica and uses the `nginx` image.

279. Kustomize Output



```
> _  
  
$ kustomize build k8s/  
  
$ kustomize build k8s/ | kubectl apply -f -  
service/nginx-loadbalancer-service created  
deployment.apps/nginx-deployment created
```

Doesn't apply/deploy config

Redirect

```
# 출력 확인하기  
kubectl kustomize ./  
# 또는  
kustomize build ./  
  
# 바로 적용하기  
kustomize build ./ | kubectl apply -f -  
  
kubectl apply -k ./  
  
# 리소스 삭제  
kustomize build ./ | kubectl delete -f -
```

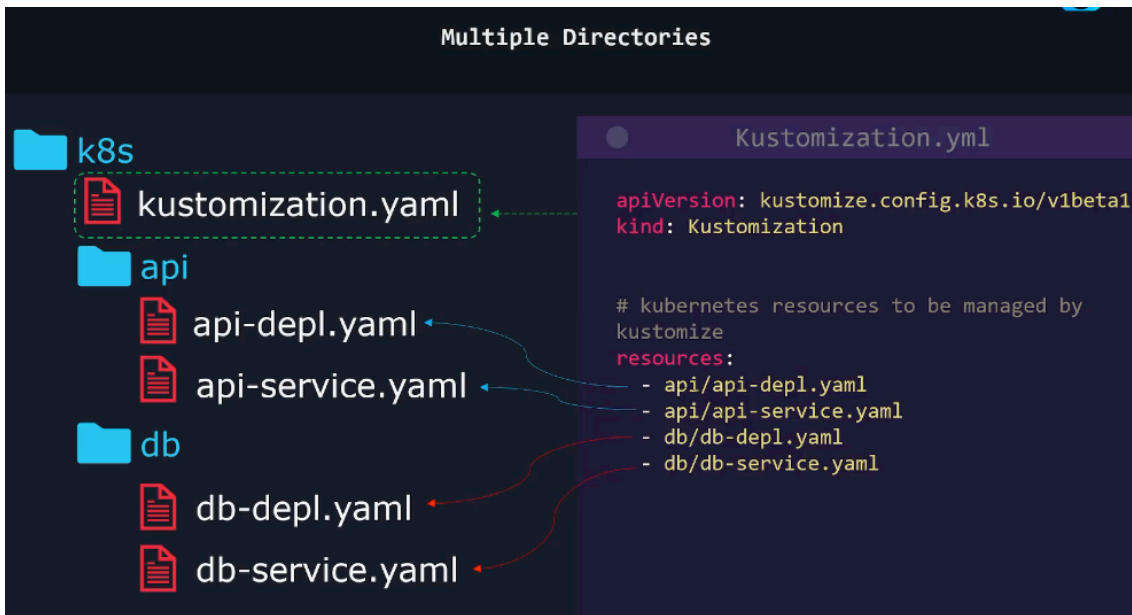
280. Kustomize ApiVersion & Kind



- 다른 쿠버네티스 리소스 파일과 마찬가지로, `kustomization.yaml` 에서 API 버전과 kind 속성을 설정할 수 있다
- 사용자 지정 가능한 기본값을 선택할 수 있지만, 향후 변경 사항이 발생할 경우를 대비하여 이러한 값을 하드 코딩하는 것이 좋다

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
```

281. Managing Directories



- 각각의 하위 디렉토리 (`k8s/api` , `k8s/db`)로 이동해서 `kubectl apply` 해도 되지만, 번거롭다.
→ `kustomization.yaml`
 - `resources:` deployment, service yaml 파일 각각의 상대 경로 지정
 - `kustomization.yaml` 정의 후, `k8s` 디렉토리 루트에서 `kustomize build k8s/ | kubectl apply -f -` 실행 : 모든 구성을 한꺼번에 apply
- 하위 디렉토리조차도 너무 많아서 `resources` 에 기입할 것이 너무 길어지면?
→ 각각의 하위 디렉토리에 `kustomization.yaml` 을 생성하고 각 디렉토리에 대한 구성만 가져오도록 설정, root의 `kustomization.yaml` 에는 각 하위 디렉토리에 대한 경로를 기입하면 됨.
루트에서 `kustomize build k8s/ | kubectl apply -f -` 실행



284. Common Transformers

- `commonLabels` : 모든 리소스에 공통 레이블 추가
- `namePrefix` / `nameSuffix` : 모든 리소스 이름에 접두사/접미사 추가
- `namespace` : 모든 리소스에 네임스페이스 설정
- `commonAnnotations` : 모든 리소스에 공통 주석 추가

```
namePrefix: dev-  
commonLabels:  
  app: myapp  
  environment: dev  
namespace: myapp-dev
```

285. Image Transformers

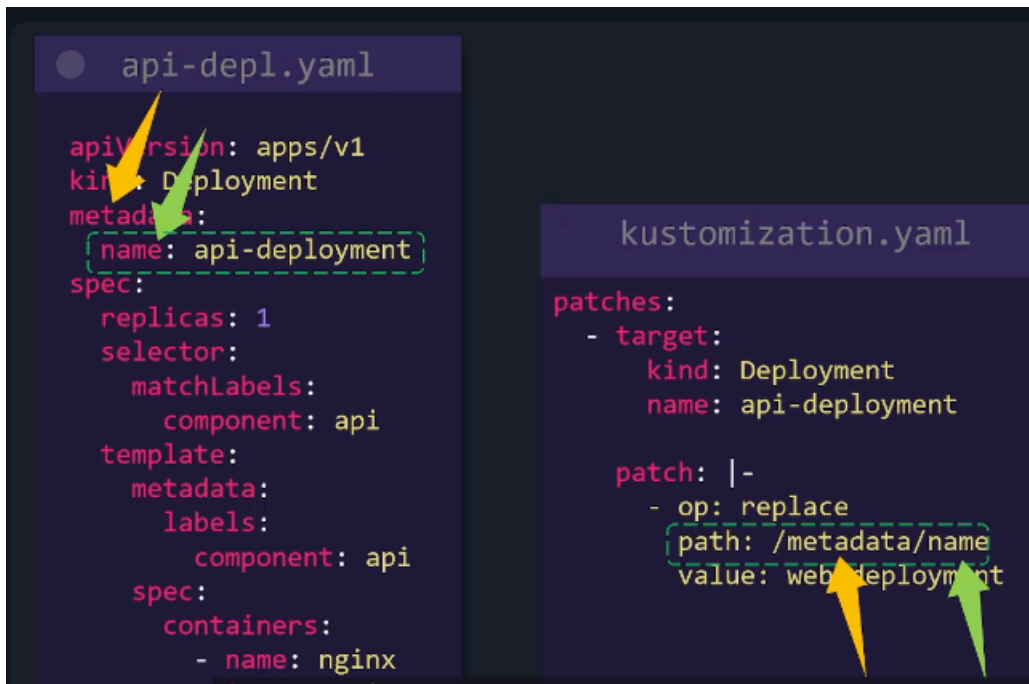
- 컨테이너 이미지 관련 필드를 변경

```
kustomization.yaml  
---  
images:  
- name: nginx  
  newName: haporxy  
  newTag: 2.4
```

- `name` : 변경할 이미지 이름
- `newName` : 새 이미지 레지스트리/이름
- `newTag` : 새 이미지 태그 (문자열)
- `digest` : 이미지 다이제스트(태그 대신 사용 가능)

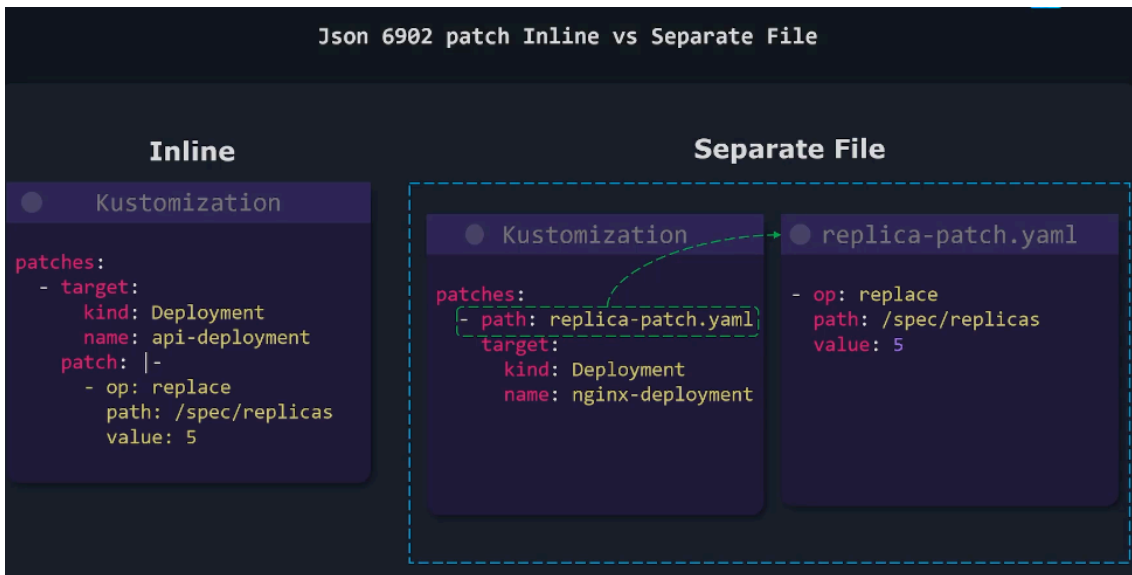
288. Patches Intro

- patch: 특정 object 하나 또는 몇 개의 object에만 적용하고 싶을 때 사용
- 3개 paramter 필요
 - operation type (작업 유형): add/remove/replace
 - target: kind, name..
 - value



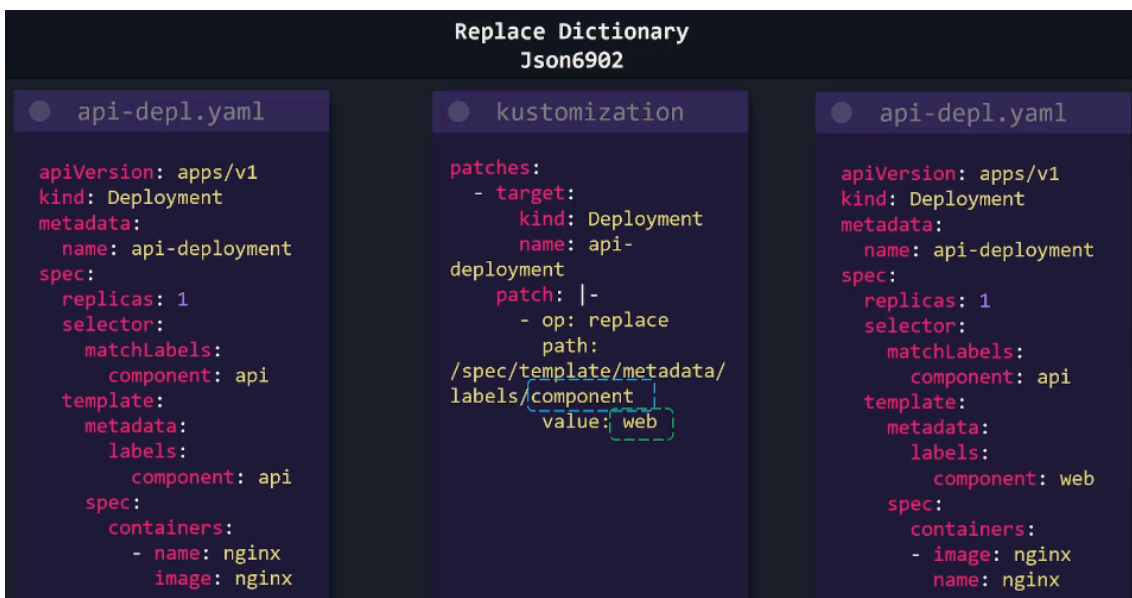
- **JSON 6902 Patch**
 - 복잡한 변경에 적합
 - `target` , `patch` (patch 세부 정보) 기입
- **Strategic Merge Patch**
 - 일반적인 쿠버네티스 config 구성과 비슷한 구조
 - 새 구성으로 이전 구성과 병합하고 변경된 내용을 파악하여 업데이트
 - 업데이트할 쿠버네티스 객체를 kustomize에 알려주고 (하늘색 표시 부분), 변경하려는 특정 property를 정의 (ex. replicas: 5)

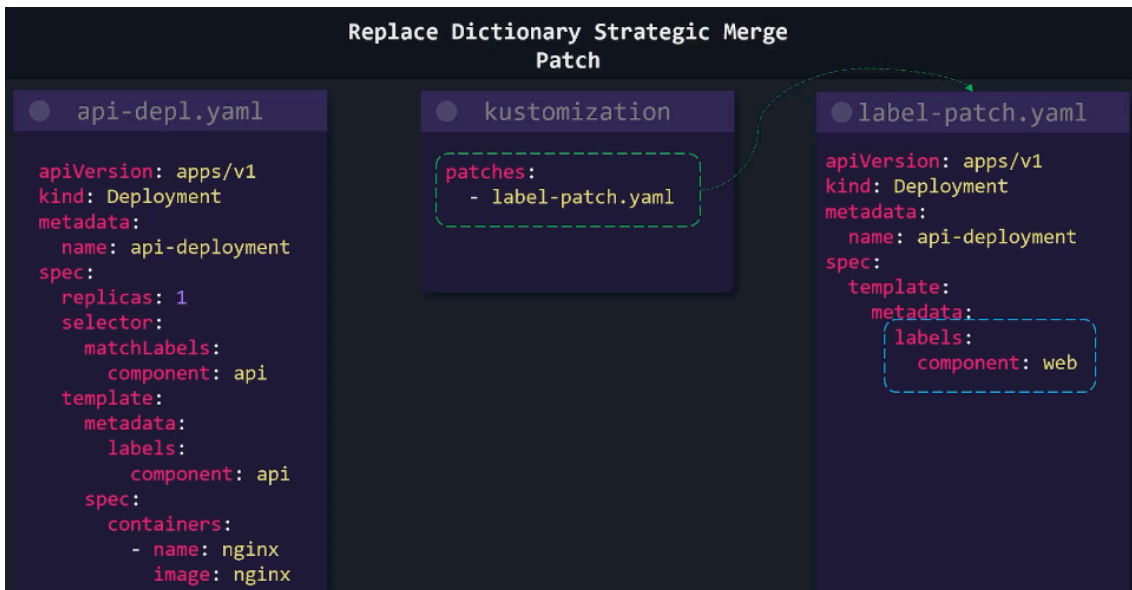
289. Different Types of Patches



- **JSON 6902 Patch** 와 **Strategic Merge Patch** 모두 patch를 정의할 수 있는 두가지 방법이 있음
 - Inline : Kustomization 내에서 patch 정의
 - separate file : 별도의 파일 사용

290. Patches Dictionary

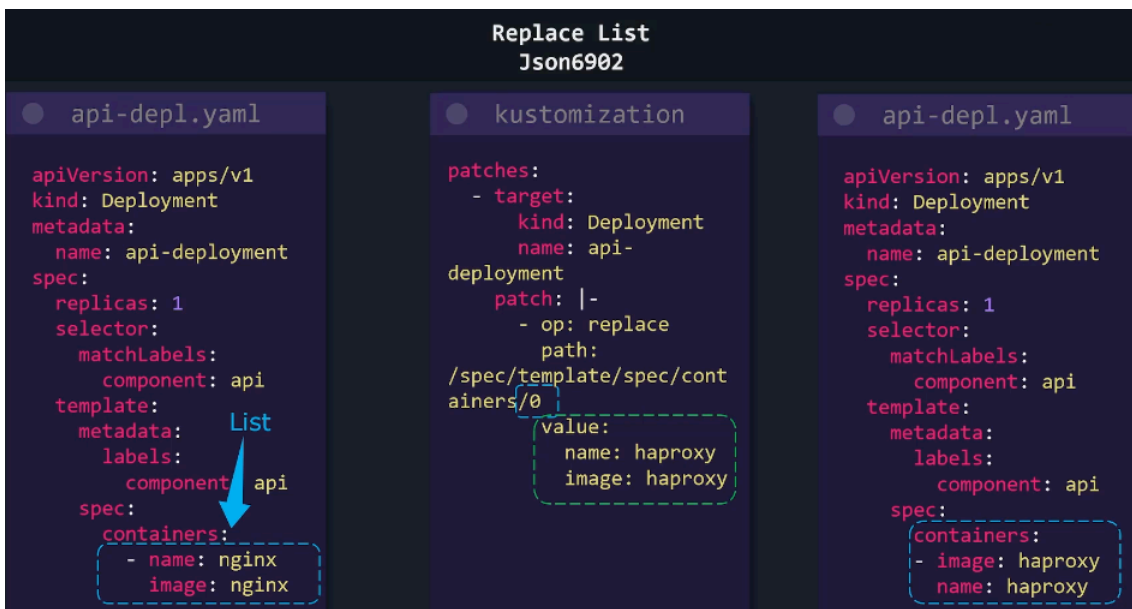




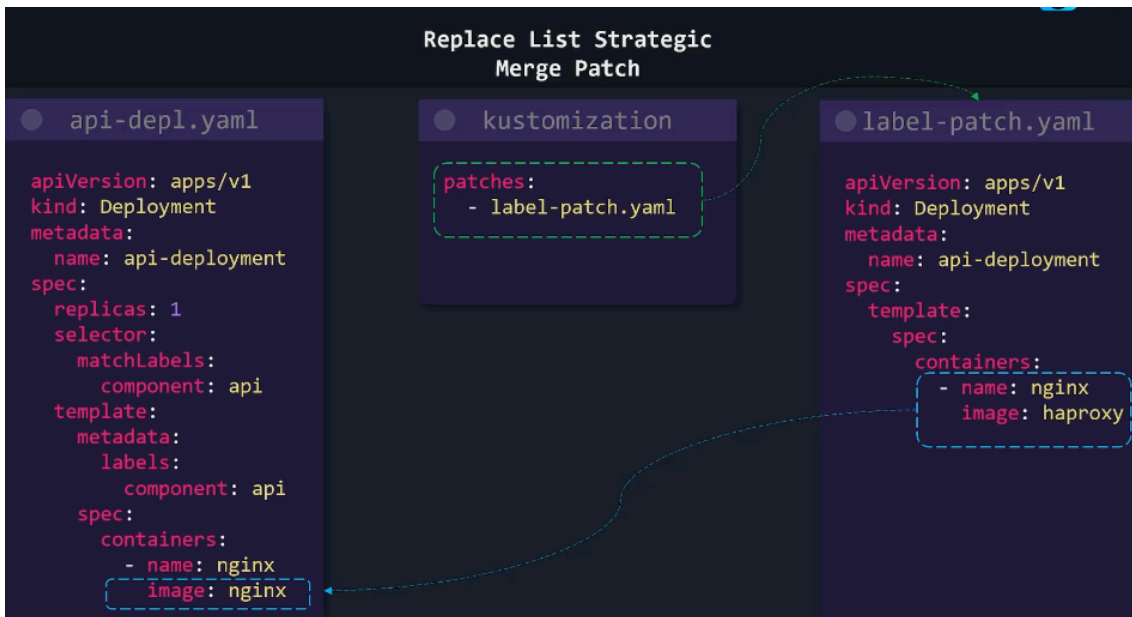
- strategic merge patch - 기존 yaml 파일 내용을 복붙한 뒤 변경되는 부분만 남기고 다 지워~
 - key 삭제하고 싶으면 null로 지정

291. Patches list

Replace List

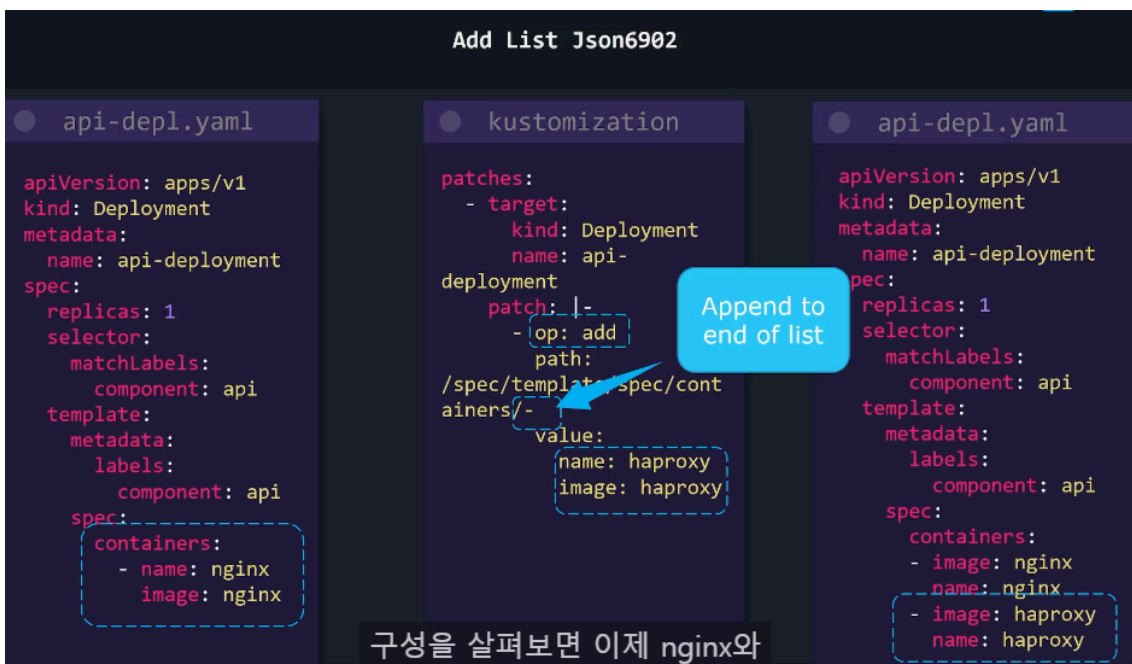


- 목록을 patch 하고 싶을 때는 path에 인덱스 숫자를 지정

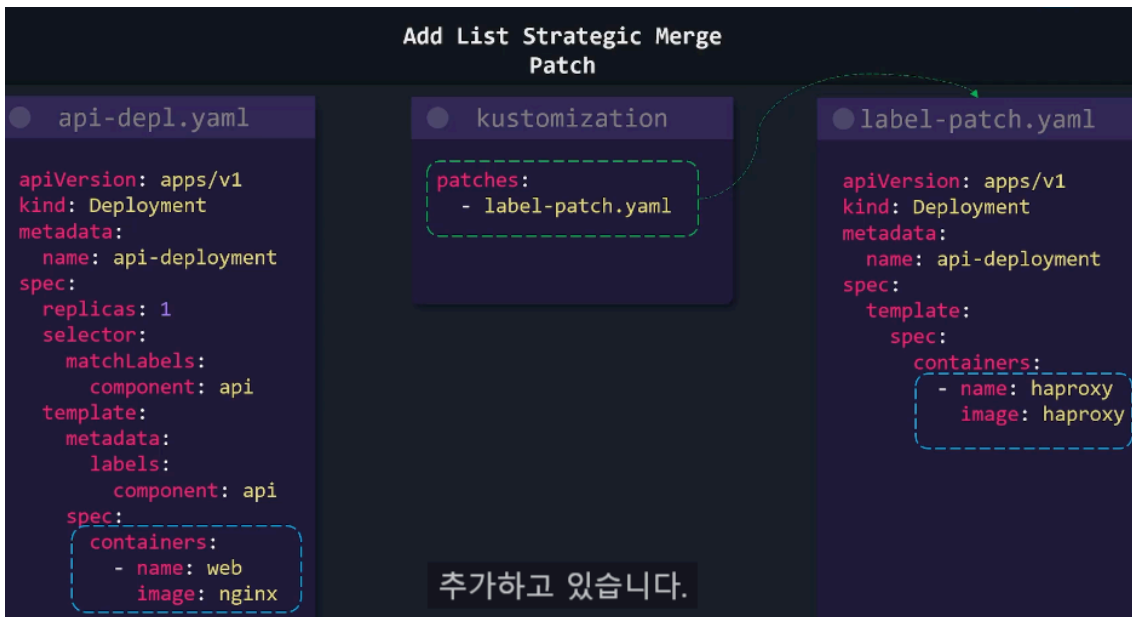


- 그냥 알던대로 하면 됨

Add List



- **path** 마지막에 **-** : 목록 끝에 추가하고 싶다는 의미
- 특정 위치에 추가하고 싶다면 인덱스로 지정 가능



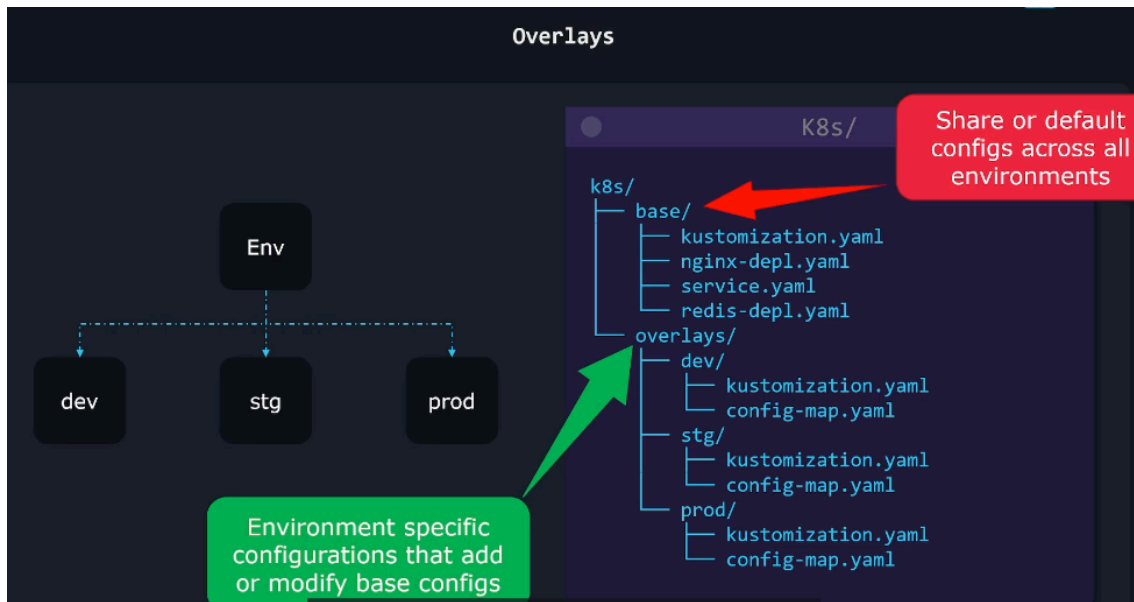
Delete List

- JSON Patch는 특이사항 없음



- Strategic Merge Patch에서는 replace, add 할 때 하듯이 yaml에 삭제된 상태를 기입하면 kustomize는 내가 아무것도 하고 싶지 않다고 생각함.
- 따라서, 삭제 지시문 사용 `$patch: delete` , 그 아래에 삭제할 내용 정확히 지정

293. Overlays



- base: 모든 환경에서 공통적인 config
- overlays: base config를 기반으로, dev, stg, prod 등 환경 별로 특정 속성을 조정
 - base config 가져오기 위해 하위 디렉토리의 `kustomization.yaml` 에 `bases: -${base 폴더에 있는 kustomization.yaml 파일 경로}` 지정
 - `resources:` 를 통해 base에 없던 새로운 리소스 추가 가능

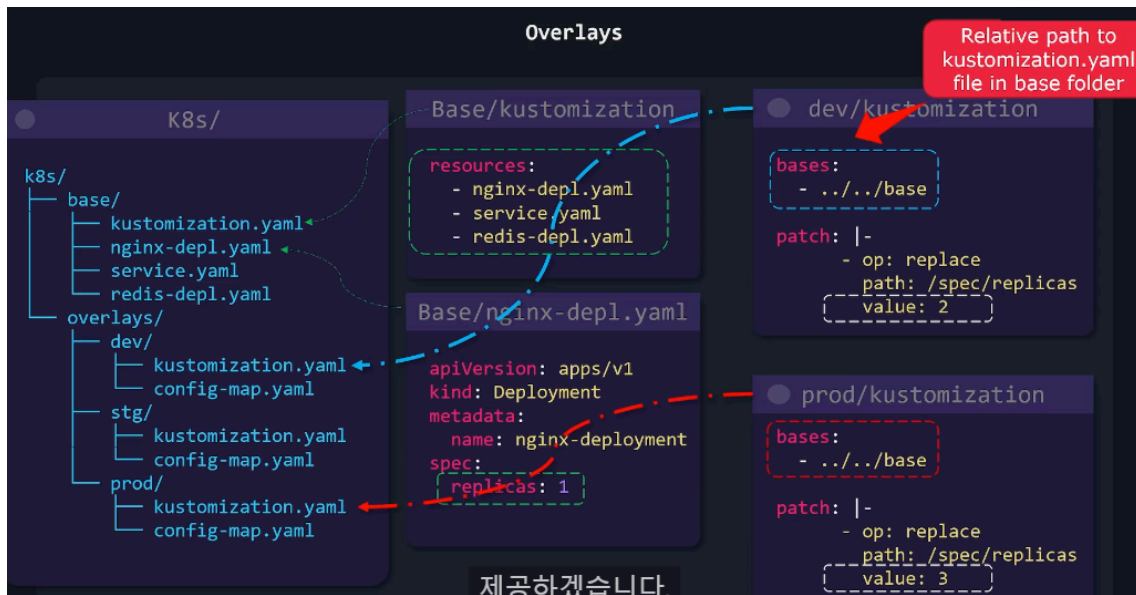
```

prod/kustomization

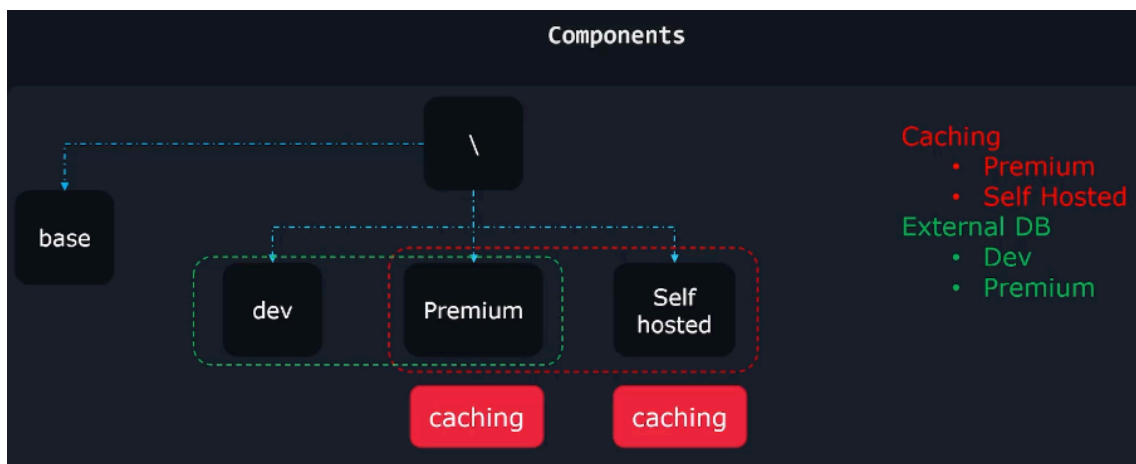
bases:
- ../../base

resources:
- grafana-depl.yaml

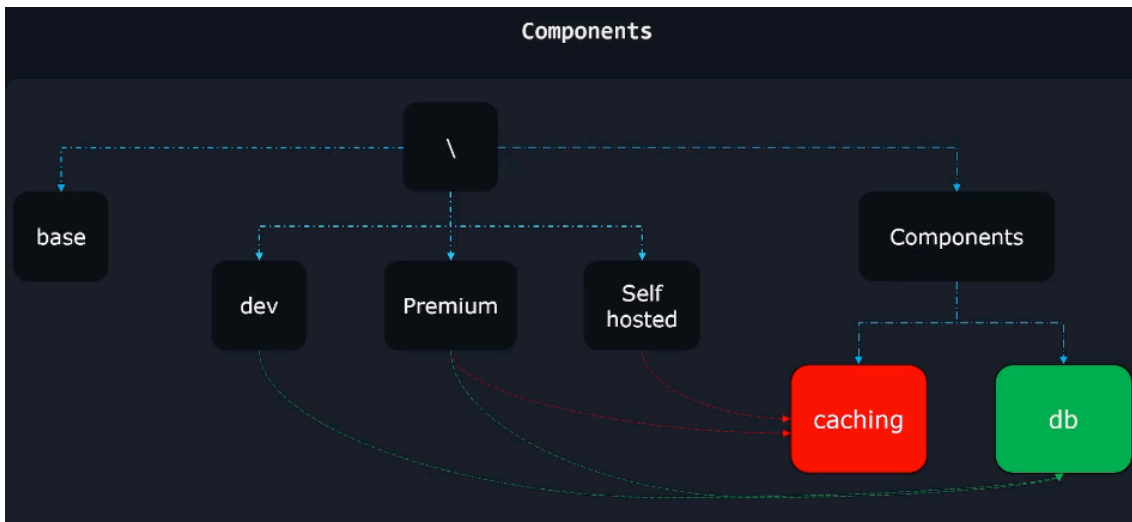
patch: |-
- op: replace
  path: /spec/replicas
  value: 2
  
```



295. Components



- 여러 overlay에 포함될 수 있는, 재사용 가능한 config logic을 정의
- overlay 하위에서 특정 기능을 선택적으로 활성화해야되는 경우 유용



1. Components 폴더에 디렉토리와 필요한 리소스 정의 후

```
# components/redis/kustomization.yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Component

resources:
- redis-deployment.yaml
- redis-service.yaml

commonLabels:
  component: redis
```

2. overlay의 `kustomization.yaml` 에서 Component import

```
# kustomization.yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- base-app.yaml

components:
- ../components/redis
- ../components/monitoring
```