쿠버네티스 플랫폼 프로그래밍

clieng-go & kubebuilder

임찬식 @ 라인플러스

Speaker

임찬식

라인플러스

- 소프트웨어 엔지니어
- 고성능 워크로드를 위한 서빙 플랫폼 개발
- Go 언어를 이용해 플랫폼 및 오퍼레이터 구현

세션에서 다룰 내용

- client-go 라이브러리로 쿠버네티스 API 사용하는 방법
- kubebuilder 프레임워크로 생성한 오퍼레이터 기본 코드 설명
- 오퍼레이터로 애플리케이션을 쿠버네티스에 배포하는 방법
- 애플리케이션 리소스를 이용한 오퍼레이터 할당 방식 소개

Index

- 쿠버네티스 API & client-go
- kubebuilder 기반 오퍼레이터 기초
- 오퍼레이터로 배포하는 애플리케이션
- 정리 및 참고자료

쿠버네티스 API & client-go

쿠버네티스 API

- 쿠버네티스 플랫폼과 상호작용할 수 있는 인터페이스
- 사용자, 관리자, 애플리케이션이

  클러스터 및 리소스를 관리하는 목적으로 사용 가능
- 리소스 목록을 얻거나 생성하는 일련의 작업 수행 가능
- 애플리케이션 배포 및 상태 모니터링 기능 제공
- HTTP API 형태로 제공되어 다양한 언어 및 도구 지원

쿠버네티스 API – 구조 (1)

https://<APISERVER>/<GROUP>/<VERSION>/<RESOURCE>/<NAME>

- <APISERVER>: API 서버 주소
- <GROUP>: 리소스 그룹
  - /api: 쿠버네티스 코어 API 그룹
  - /apis/*: 쿠버네티스 확장 API 그룹 (예: /apis/apps)

쿠버네티스 API – 구조 (2)

https://<APISERVER>/<GROUP>/<VERSION>/<RESOURCE>/<NAME>

- <VERSION>: v1(안정 버전), v1beta1, v1alpha1
- <RESOURCE>: 접근하려는 리소스 종류 (/pods, /services)
- <NAME>: 접근하려는 리소스 이름 지정

쿠버네티스 API – 구조: NAMESPACE 지정

.../<VERSION>/namespaces/<NAMESPACE><RESOURCE>/<NAME>

- <NAMESPACE>: 리소스가 속한 네임스페이스 이름
- 네임스페이스 범위 리소스를 네임스페이스 조건을 빼고 조회하면 모든 네임스페이스에서 검색
  - 단 접근 권한을 가진 네임스페이스만 조회 가능

쿠버네티스 API - 그룹 정리

- 코어(Core) 그룹
  - /api/<version> 경로로 접근 가능
  - 기본적이고 필수적인 리소스: Pod, Service 등
- 확장 그룹
  - /apis/<group>/<version> 경로로 접근 가능
  - 애플리케이션 배포 및 확장 기능에 관계된 리소스

쿠버네티스 API – CURL 호출 (1)

- curl 명령을 이용해 HTTP API 엔드포인트 호출 가능
- https 연결을 위해 인증서 준비

```
server=$(grep 'server' ~/.kube/config | awk '{print $2}')
grep 'certificate-authority' ~/.kube/config | awk '{print $2}' | base64 --decode > cacert
grep 'client-certificate' ~/.kube/config | awk '{print $2}' | base64 --decode > cert
grep 'client-key' ~/.kube/config | awk '{print $2}' | base64 --decode > key
```

- curl 명령을 실행할 때 추출한 인증서 파일 사용

쿠버네티스 API – CURL 호출 (2)

- curl 명령을 이용해 Pods 목록 조회

```
curl --cacert cacert --cert cert --key key \
  "$server/api/v1/namespaces/defaults/pods”
```

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "1436678”
  },
  "items": [...]
}
```

쿠버네티스 API – CURL 호출 (3)

- curl 명령을 이용해 Deployments 목록 조회

```
curl --cacert cacert --cert cert --key key
  "$server/apis/apps/v1/namespaces/defaults/deployments"
{
  "kind": "DeploymentList",
  "apiVersion": "apps/v1",
  "metadata": {
    "resourceVersion": "1508635"
  },
  "items": []
}
```

client-go 라이브러리 특징

- 쿠버네티스 API 호출을 추상화해 사용하기 편하게 제공
- 쿠버네티스 코어 그룹과 확장 그룹 모두 접근 가능
  - 기본 리소스 외 사용자 정의 리소스 접근 및 사용 가능
- 클러스터 내부/외부에서 초기화 기능 지원
- 리소스 변경 사항을 캐싱하는 Informers 등 다양한 기능 지원

Pods 목록을 조회하는 Go 프로그램 (1)

- client-go 패키지와 Meta 정보를 표현하는 패키지 사용

```
package main
import (
    "context"
    "fmt"
    "os"
    "path/filepath"

    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes"
    "k8s.io/client-go/tools/clientcmd"
)
```

Pods 목록을 조회하는 Go 프로그램 (2)

- .kube/config 파일에 저장된 클러스터 정보를 사용해 초기화

```
func main() {
    homeDir, err := os.UserHomeDir()
    if err != nil {
        panic(err)
    }

    kubeConfig := filepath.Join(homeDir, ".kube", "config")
    config, err := clientcmd.BuildConfigFromFlags("", kubeConfig)
    if err != nil {
        panic(err)
    }
```

Pods 목록을 조회하는 Go 프로그램 (3)

- 예제 프로그램은 클러스터 밖에서 접근하는 경우를 가정
- .kube/config 파일에 정의한 current-context 정보 사용
- 쿠버네티스에 접근할 수 있는 clientSet 생성

```
clientSet, err := kubernetes.NewForConfig(config)
if err != nil {
    panic(err)
}
```

Pods 목록을 조회하는 Go 프로그램 (4)

- Core 그룹에 속한 Pods 리소스 목록 조회

```
    pods, err := clientSet.CoreV1().
        Pods("default").
        List(context.TODO(), metav1.ListOptions{})
    if err != nil {
        panic(err.Error())
    }

    fmt.Printf("There are %d pods.\n", len(pods.Items))
    for _, pod := range pods.Items {
        fmt.Printf("Pod Name: %s\n", pod.Name)
    }
}
```

Pods 목록을 조회하는 Go 프로그램 (5)

- 프로그램을 실행해 default 네임스페이스의 Pods 목록 확인

```
$ go run get-pods.go
There are 1 pods.
Pod Name: nginx-9d4c4bf9-8pxt6
```

- Pods(“”) 호출을 통해 모든 네임스페이스 조회 가능
  - 접근 권한을 가진 네임스페이스 한정

모든 Deployments 목록 조회 (1)

- 모든 네임스페이스의 Deployments 목록 조회
- 네임스페이스에 “” 문자열을 전달

```
deployments, _ := clientSet.AppsV1().
    Deployments("").
    List(context.Background(), metav1.ListOptions{})

for _, item := range deployments.Items {
    fmt.Printf("Deployment: %s/%s, %d Replicas\n",
        item.Namespace, item.Name, item.Status.Replicas)
}
```

모든 Deployments 목록 조회 (2)

- 예제에서는 관리자 권한을 이용해 테스트 진행

```
$ go run get-deployments.go
Deployment: default/nginx, 1 Replicas
Deployment: kube-system/cilium-operator, 1 Replicas
Deployment: kube-system/coredns, 2 Replicas
```

- Deployment 리소스 Spec, Status 정보에 접근 가능
- 일반적으로 Status 속성은 읽기만 가능하도록 권한이 부여됨

Deployment 컨테이너 이미지 수정 (1)

- Deployment 리소스에 정의한 컨테이너 이미지 확인

```
ctx := context.Background()
deployment, _ := clientSet.AppsV1().
    Deployments("default").
    Get(ctx, "nginx", metav1.GetOptions{})
```

```
name := deployment.GetName()
image := deployment.Spec.Template.Spec.Containers[0].Image
fmt.Printf("Deployment: %s, Container: %s\n", name, image)
```

Deployment 컨테이너 이미지 수정 (2)

- 읽은 Deployment 리소스 컨테이너의 이미지 필드 변경

```
deployment.Spec.Template.Spec.Containers[0].Image = "nginx:1.26"
deployment, err = clientSet.AppsV1().
    Deployments("default").
    Update(ctx, deployment, metav1.UpdateOptions{})
if err != nil {
    fmt.Println(err)
}

// 이미지를 변경한 후 파드 배포가 진행되는 동안 대기.
time.Sleep(15 * time.Second)
```

Deployment 컨테이너 이미지 수정 (3)

- 새로 변경된 Pod 리소스 목록을 읽어 변경된 이미지 확인

```
labelSelector :=
    labels.SelectorFromSet(deployment.Spec.Selector.MatchLabels)
pods, _ := clientSet.CoreV1().
    Pods("default").
    List(ctx, metav1.ListOptions{
        LabelSelector: labelSelector.String(),
    })
name = pods.Items[0].Name
image = pods.Items[0].Spec.Containers[0].Image
fmt.Printf("Pod: %s, Image: %s\n", name, image)
```

Deployment 컨테이너 이미지 수정 (4)

- Deployment 리소스를 변경해 원하는 이미지로 Pod 배포

```
$ go run update-deployment.go
Deployment: nginx, Container: nginx:latest
Pod: nginx-79cc74f48c-svbfm, Image: nginx:1.26
```

- 처음 Deployment 리소스를 읽을 때 이미지는 nginx:latest
- Deployment 리소스를 변경하면
  기존 Pod 삭제 후 nginx:1.26 이미지를 가진 Pod 배포

동시에 리소스를 변경한 경우 ? (1)

- Deployment 리소스를 동시에 변경하는 경우
- 가장 먼저 도착한 변경 사항은 적용되고 나머지는 실패

```
$ go run update-deployment.go
Deployment: nginx, Container: nginx:latest
Operation cannot be fulfilled on deployments.apps "nginx":
the object has been modified;
please apply your changes to the latest version and try again
```

동시에 리소스를 변경한 경우 ? (2)

- 서로 다른 서버에서 동시에 이미지를 변경하려고 시도

```
deployment.Spec.Template.Spec.Containers[0].Image = "nginx:1.26"
deployment, err = clientSet.AppsV1().
    Deployments("default").
    Update(ctx, deployment, metav1.UpdateOptions{})
```

```
deployment.Spec.Template.Spec.Containers[0].Image = "nginx:1.27"
deployment, err = clientSet.AppsV1().
    Deployments("default").
    Update(ctx, deployment, metav1.UpdateOptions{})
```

동시에 리소스를 변경한 경우 ? (3)

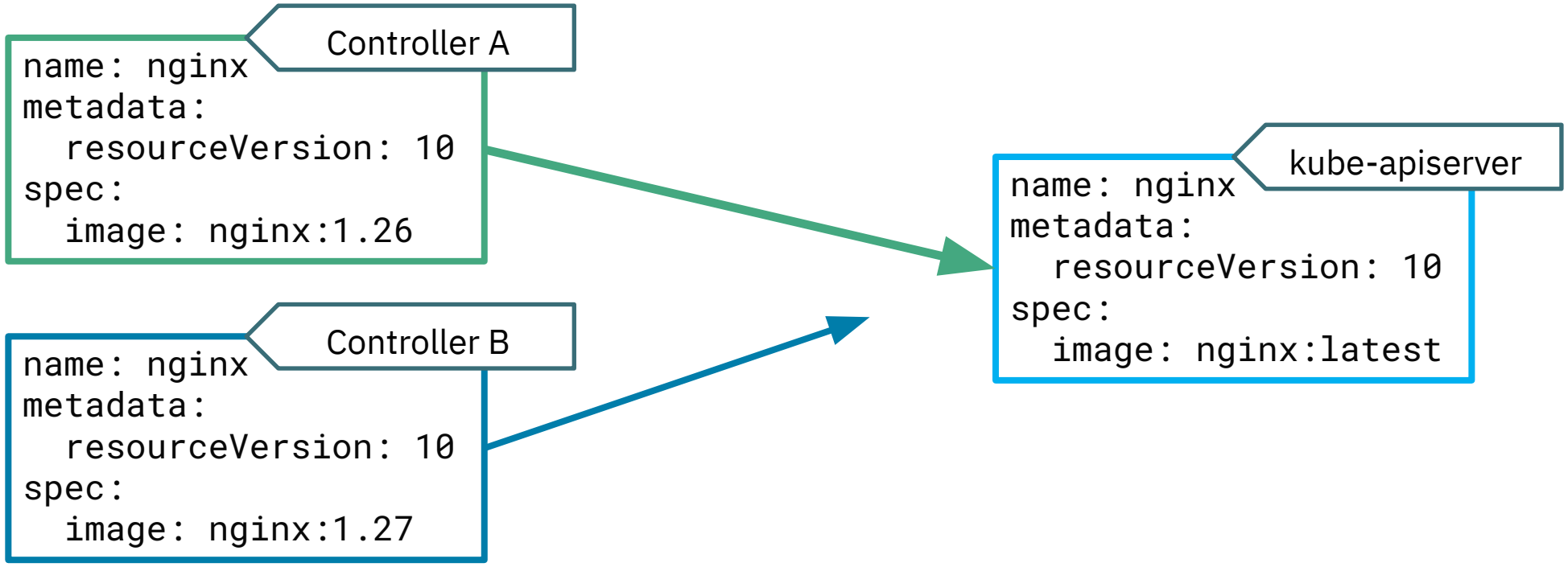
- 한 요청이 성공하면 다른 요청은 Conflict 에러와 함께 실패

```
deployment.Spec.Template.Spec.Containers[0].Image = "nginx:1.26"
deployment, err = clientSet.AppsV1().
    Deployments("default").
    Update(ctx, deployment, metav1.UpdateOptions{})
```

```
deployment.Spec.Template.Spec.Containers[0].Image = "nginx:1.27"
deployment, err = clientSet.AppsV1().
    Deployments("default").
    Update(ctx, deployment, metav1.UpdateOptions{})
```

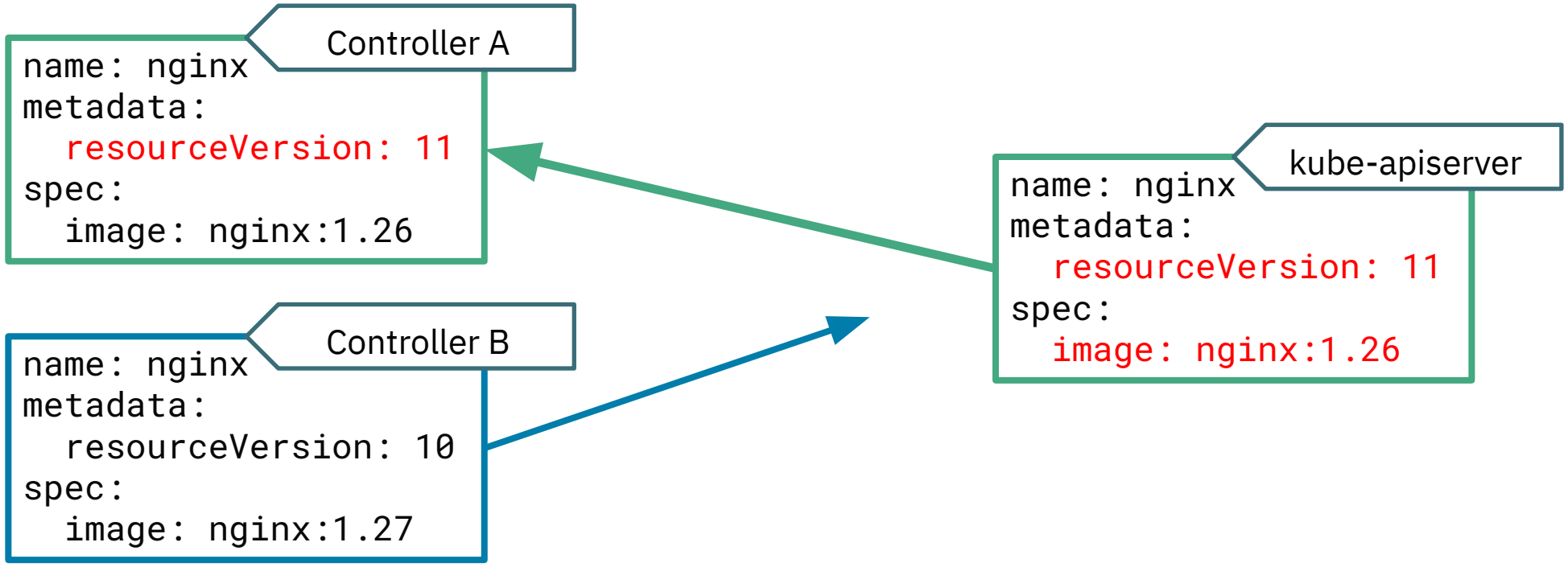
리소스 버전에 기반한 낙관적 동시성 (1)

- 변경 요청을 전송하면 리소스 버전을 이용해 충돌 감지



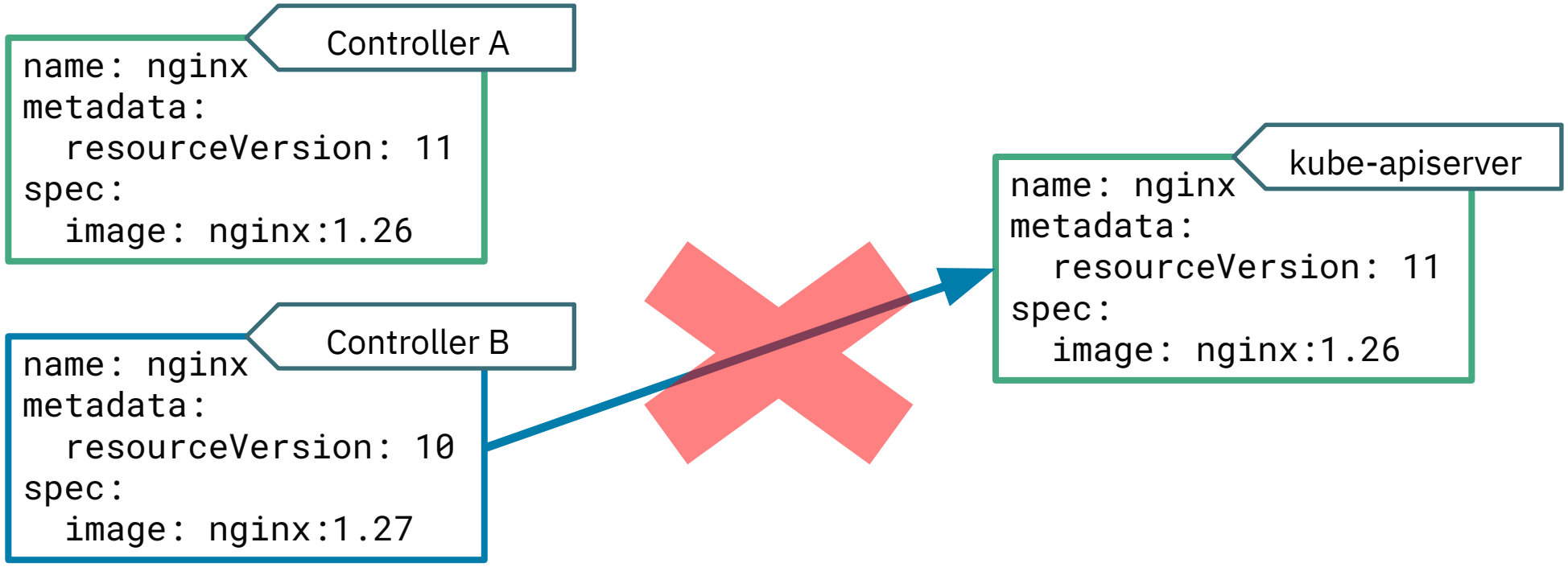
리소스 버전에 기반한 낙관적 동시성 (2)

- 저장된 리소스 버전과 동일한 버전을 가진 변경 사항을 적용



리소스 버전에 기반한 낙관적 동시성 (3)

- 리소스 버전이 다른 수정 사항일 경우 충돌로 인해 실패



수정할 때 충돌이 발생할 경우 처리 (1)

- 변경할 리소스를 읽은 후 리소스 필드 변경
- 리소스를 변경이 모두 끝난 후 Update() 함수 호출
- 충돌이 나서 실패할 경우, 리소스를 다시 읽은 후 변경
- retry.RetryOnConflict() 함수를 이용해 간단하게 사용 가능

수정할 때 충돌이 발생할 경우 처리 (2)

- 내부 함수는 변경할 리소스를 읽어오면서 시작

```
err = retry.RetryOnConflict(retry.DefaultRetry, func() error {
    newDeployment, getErr := clientSet.AppsV1().
       Deployments("default").
       Get(ctx, "nginx", metav1.GetOptions{})
    if getErr != nil {
       return getErr
    }
    newDeployment.Spec.Template.Spec.Containers[0].Image = "nginx:1.27"
    _, updateErr := clientSet.AppsV1().
       Deployments("default").
       Update(ctx, newDeployment, metav1.UpdateOptions{})
    if updateErr != nil {
        fmt.Printf("Update Deployment Error: %v", updateErr)
    }
    return updateErr
})
```

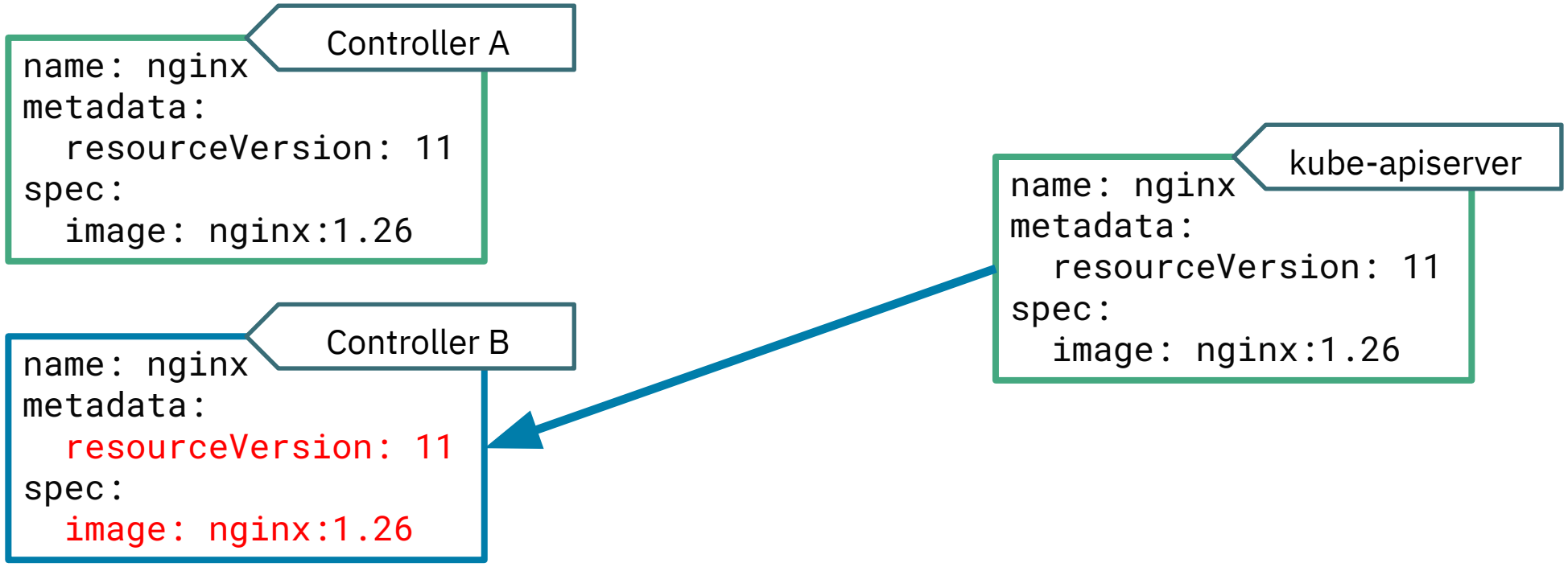
수정할 때 충돌이 발생할 경우 처리 (3)

- 리소스를 변경하고 Update() 호출 후 에러 처리

```
err = retry.RetryOnConflict(retry.DefaultRetry, func() error {
    newDeployment, getErr := clientSet.AppsV1().
        Deployments("default").
        Get(ctx, "nginx", metav1.GetOptions{})
    if getErr != nil {
        return getErr
    }
    newDeployment.Spec.Template.Spec.Containers[0].Image = "nginx:1.27"
    _, updateErr := clientSet.AppsV1().
        Deployments("default").
        Update(ctx, newDeployment, metav1.UpdateOptions{})
    if updateErr != nil {
        fmt.Printf("Update Deployment Error: %v", updateErr)
    }
    return updateErr
})
```

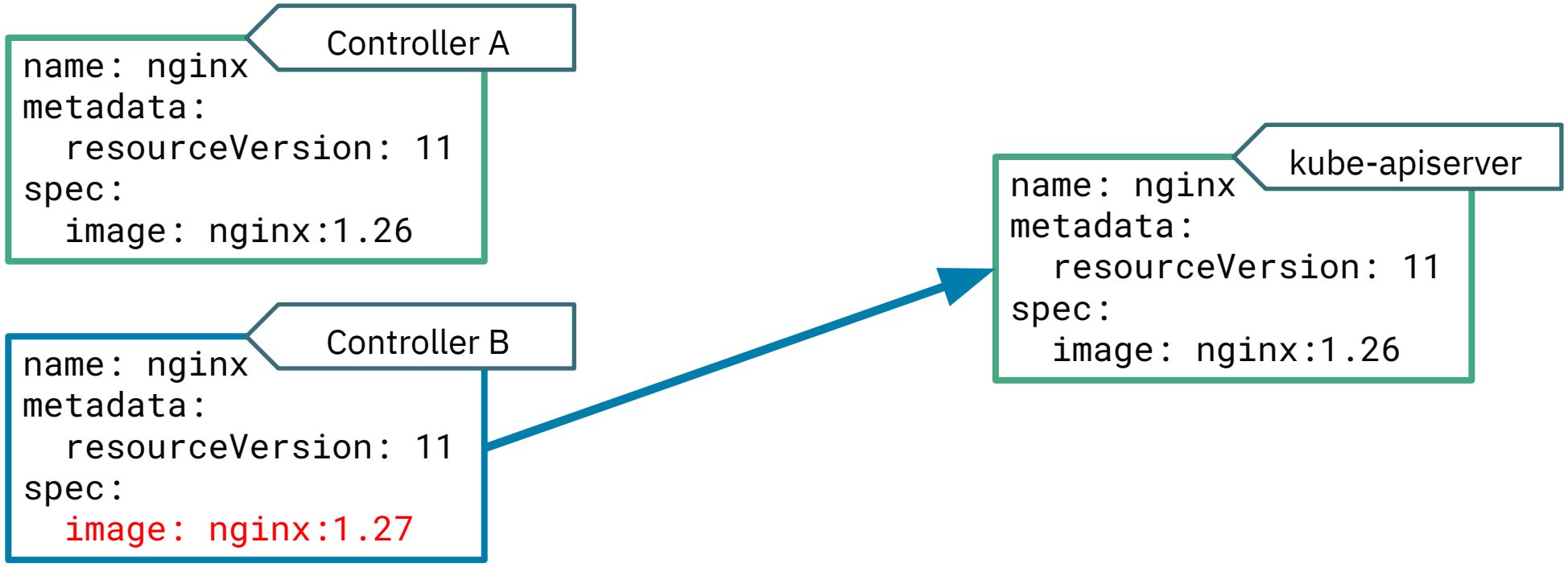
수정할 때 충돌이 발생할 경우 처리 (4)

- Update() 함수가 충돌로 실패하면 리소스를 다시 읽음



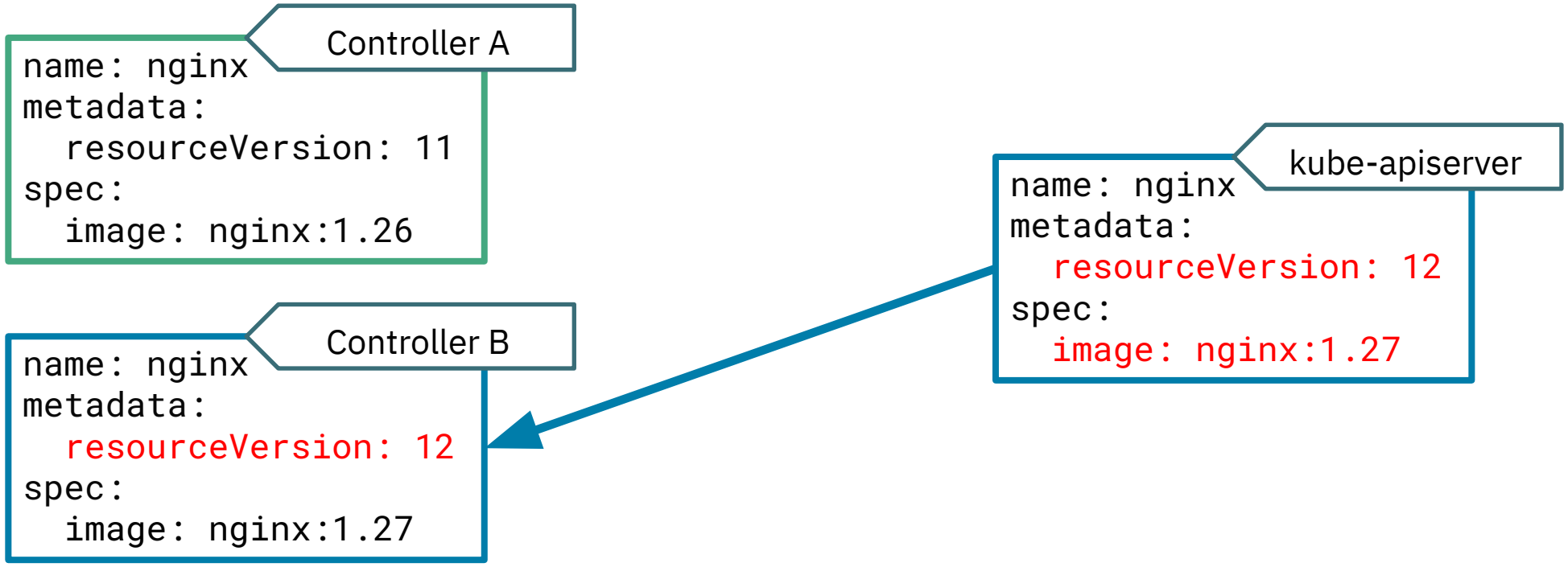
수정할 때 충돌이 발생할 경우 처리 (5)

- 읽은 최신 리소스에 원하는 변경 사항 적용 후 Update()



수정할 때 충돌이 발생할 경우 처리 (6)

- 동일한 리소스 버전을 가진 변경 요청이기 때문에 성공



수정할 때 충돌이 발생할 경우 처리 (7)

- 가장 마지막에 변경한 내용으로 리소스 정보 갱신

Controller A

```
name: nginx
metadata:
  resourceVersion: 11
spec:
  image: nginx:1.26
```

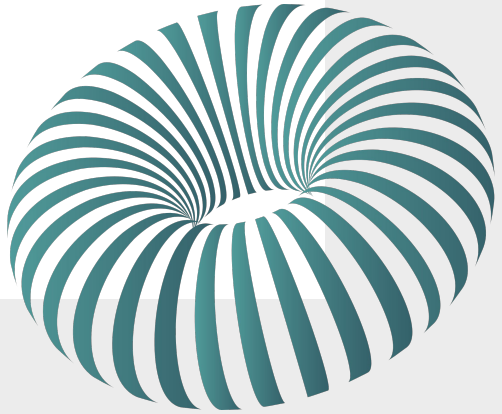
Controller B

```
name: nginx
metadata:
  resourceVersion: 12
spec:
  image: nginx:1.27
```

kube-apiserver

```
name: nginx
metadata:
  resourceVersion: 12
spec:
  image: nginx:1.27
```

kubebuilder 기반 오퍼레이터 기초



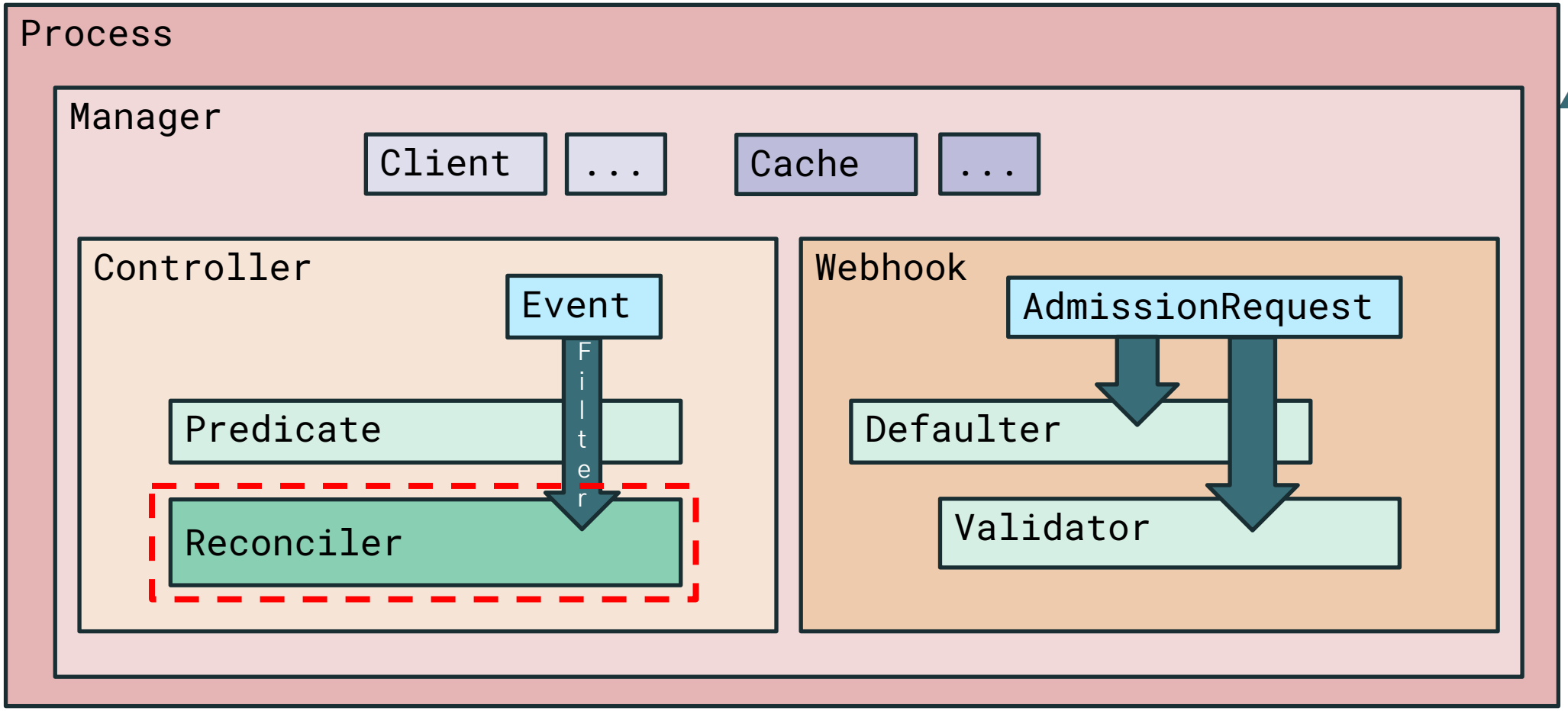
쿠버네티스 컨트롤러와 오퍼레이터

- 컨트롤러
  - 쿠버네티스 기본 리소스를 주로 관리하는 목적
  - Pod, Deployment, Service 등의 리소스 관리
- 오퍼레이터
  - 사용자 정의 리소스를 통해 복잡한 상태 관리 가능
  - 애플리케이션 배포 및 관리 자동화를 위해 사용

kubebuilder 프레임워크

- 쿠버네티스 컨트롤러 혹은 오퍼레이터를 개발하기 위해 도움을 주는 Go 언어 기반 프레임워크
- 사용자 리소스 관리 스크립트 및 컨트롤러 기본 코드 제공
- 오퍼레이터를 통해 관리하려는 대상에 집중할 수 있음
- client-go 라이브러리를 이용해 개발하는 것도 가능
  - 세부적인 제어를 할 수 있지만, 매우 복잡한 작업 필요

kubebuilder 구조 (1)



https://kubebuilder.io/architecture

kubebuilder 구조 (2)

- Process: 애플리케이션 설정 및 초기화
- Manager: API 서버와 통신, 이벤트 캐시, 메트릭 노출, …
- Webhook: 리소스 초기화, 데이터 검증을 진행할 웹 훅 설정
- Controller: 리소스 이벤트를 필터링해 Reconciler 호출
- Reconciler: 사용자 리소스를 관리하는 실질적인 로직 구현

kubebuilder 프로젝트 생성 (1)

- kubebuilder init 명령으로 프로젝트 기본 파일 생성

```
$ mkdir project
$ cd project
$ kubebuilder init --domain tutorial --repo tutorial/project
INFO Writing kustomize manifests for you to edit...
INFO Writing scaffold for you to edit...
INFO Get controller runtime:
$ go get sigs.k8s.io/controller-runtime@v0.19.0
INFO Update dependencies:
$ go mod tidy
Next: define a resource with:
$ kubebuilder create api
```

kubebuilder 프로젝트 생성 (2)

- kubebuilder create api 명령으로 CronJob 리소스 생성

```
$ kubebuilder create api --group batch --version v1 --kind CronJob
INFO Create Resource [y/n] y
INFO Create Controller [y/n] y
INFO Writing kustomize manifests for you to edit...
INFO Writing scaffold for you to edit...
INFO api/v1/cronjob_types.go
INFO api/v1/groupversion_info.go
INFO internal/controller/cronjob_controller.go
INFO internal/controller/cronjob_controller_test.go
...
Next: implement your new API and generate the manifests with:
$ make manifests
```

kubebuilder 프로젝트 디렉토리 (1)

- 프로젝트 초기 코드 및 CronJob 리소스까지 생성
- api/v1: CronJob 리소스 속성을 정의하는 파일들
- cmd: 애플리케이션 시작을 담당하는 main.go 파일
- config: 애플리케이션 배포에 사용되는 다양한 설정 파일들
- internal: 실제 CronJob 컨트롤러 기본 코드 및 테스트

kubebuilder 프로젝트 디렉토리 (2)

- api/v1/cronjob_types.go: 리소스 Spec & Status 정의

```go
type CronJobSpec struct {
    // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
    // Important: Run "make" to regenerate code after modifying this file
}

// CronJobStatus defines the observed state of CronJob
type CronJobStatus struct {
    // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
    // Important: Run "make" to regenerate code after modifying this file
}
```

kubebuilder 프로젝트 디렉토리 (3)

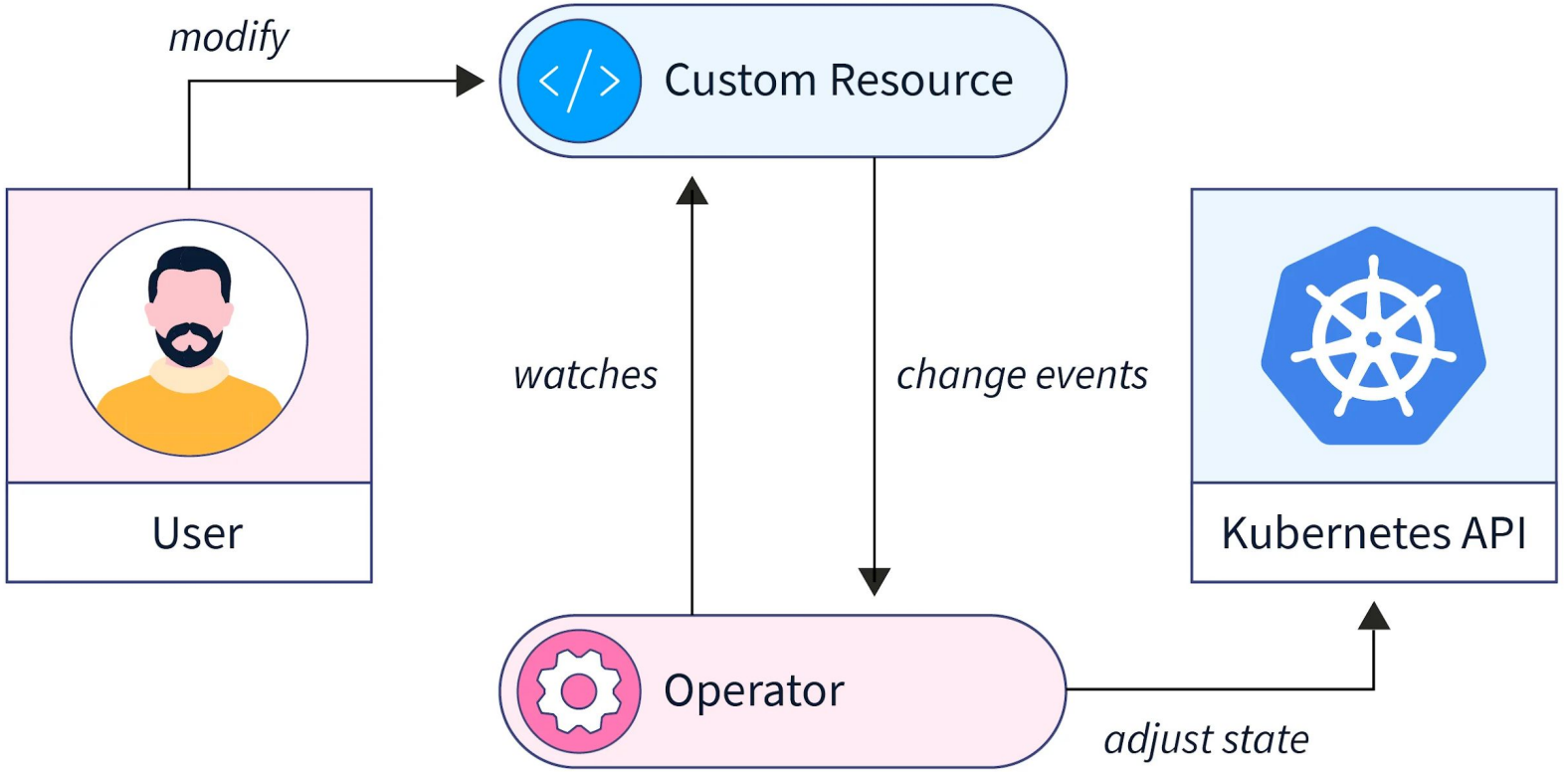
- internal/controller/cronjob_controller.go:
  CronJob 리소스에 이벤트가 발생할 때 처리할 함수 구현

```
func (r *CronJobReconciler) Reconcile(
    ctx context.Context, req ctrl.Request,
) (ctrl.Result, error) {
    _ = log.FromContext(ctx)

    // TODO(user): your logic here

    return ctrl.Result{}, nil
}
```

리소스 Spec & Status 정의 (1)

- 리소스 성격에 맞게 Spec 및 Status 정의
- Spec
  - 리소스 특징 및 원하는 상태(Desired State) 지정
- Status
  - 리소스 현재 상태(Current State) 표현

리소스 Spec & Status 정의 (2)



https://www.scaler.com/topics/kubernetes/kubernetes-operator/

리소스 Spec & Status 정의 (3)

- 리소스 변경 후 CRD 및 코드 생성 과정 진행 필요
- make manifests
  - Custom Resource Definition 파일 생성
- make generate
  - controller-gen 프로그램으로 리소스 복사 코드 생성

리소스 Spec & Status 정의 (4)

- config/crd/bases/batch.tutorial_cronjobs.yaml

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.16.1
  name: cronjobs.batch.tutorial
spec:
  group: batch.tutorial
  names:
    kind: CronJob
    listKind: CronJobList
    plural: cronjobs
    singular: cronjob
```

리소스 Spec & Status 정의 (5)

- api/v1/zz_generated.deepcopy.go

```
func (in *CronJob) DeepCopy() *CronJob {
    if in == nil {
        return nil
    }
    out := new(CronJob)
    in.DeepCopyInto(out)
    return out
}
...
```

컨트롤러 옵션 설정 (1)

- 리소스가 변경될 때 호출되는 Reconcile() 함수 관련 설정
- 변경 감지 리소스, 필터, 성능 옵션 등 다양한 설정 변경 가능

```
func (r *WeatherReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&weatherv1alpha1.Weather{}).
        WithEventFilter(predicate.GenerationChangedPredicate{}).
        WithOptions(controller.Options{
            MaxConcurrentReconciles: 1,
        }).
        Complete(r)
}
```

컨트롤러 옵션 설정 (2)

- Spec 속성이 변경될 때만 Reconcile() 함수가 호출되도록 Generation 필드를 사용하는 이벤트 필터 설정

```
func (r *WeatherReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&weatherv1alpha1.Weather{}).
        WithEventFilter(predicate.GenerationChangedPredicate{}).
        WithOptions(controller.Options{
            MaxConcurrentReconciles: 1,
        }).
        Complete(r)
}
```

컨트롤러 옵션 설정 (3)

- Reconcile() 함수가 동시에 몇 개까지 실행될 수 있는지 컨트롤러 옵션을 이용해 설정

```
func (r *WeatherReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&weatherv1alpha1.Weather{}).
        WithEventFilter(predicate.GenerationChangedPredicate{}).
        WithOptions(controller.Options{
            MaxConcurrentReconciles: 1,
        }).
        Complete(r)
}
```

Reconcile() 함수가 호출되는 상황

- 처음 오퍼레이터가 실행되는 경우
  - 모든 리소스를 순회하면서 Reconcile() 함수 호출
- 리소스가 변경되는 경우
  - 일반적으로는 리소스 필드가 변경되면 호출
  - Spec 속성이 변경될 때만 호출되도록 설정 가능

Reconcile() 함수 반환 값 제어 (1)

- Reconcile() 함수 반환 값을 이용해 주기적인 호출 제어 가능

```
func (r *CronJobReconciler) Reconcile(
    ctx context.Context, req ctrl.Request,
) (ctrl.Result, error) {
    _ = log.FromContext(ctx)

    // TODO(user): your logic here

    return ctrl.Result{
        Requeue:      false,
        RequeueAfter: 0,
    }, nil
}
```

Reconcile() 함수 반환 값 제어 (2)

- Requeue: Reconcile() 함수 재호출 여부 결정
- RequeueAfter: 일정한 시간 이후에 재호출되도록 설정 가능
- error: Reconcile() 내부에서 발생한 에러로 인해 재호출

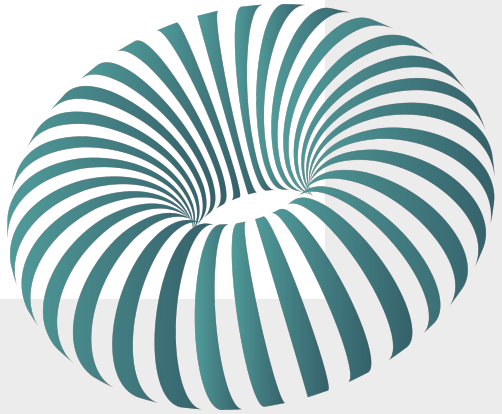
```
    return ctrl.Result{
        Requeue:      false,
        RequeueAfter: 0,
    }, nil
```

Reconcile() 함수 반환 값 제어 (3)

- 매 1분마다 Reconcile() 함수가 호출되도록 반환 값 설정

```
func (r *CronJobReconciler) Reconcile(
    ctx context.Context, req ctrl.Request,
) (ctrl.Result, error) {
    _ = log.FromContext(ctx)

    // TODO(user): your logic here

    return ctrl.Result{
        Requeue:      true,
        RequeueAfter: 1 * time.Minute,
    }, nil
}
```

오퍼레이터로 배포하는 애플리케이션



쿠버네티스에 애플리케이션 배포 (1)

- 애플리케이션을 컨테이너로 묶는 작업 필요
  - 컨테이너 생성을 위해 Dockerfile 작성
  - docker build & docker push 작업 진행
- 쿠버네티스 배포를 위해 다양한 리소스 생성
  - Deployment, ConfigMap, Secret, Service 등

쿠버네티스에 애플리케이션 배포 (2)

- Helm 차트를 이용해 배포하는 방법
  - Helm 템플릿으로 애플리케이션에 필요한 리소스 작성
- 오퍼레이터를 이용해 배포하는 방법
  - 사용자 리소스로 Spec 속성을 정의해 배포
  - Spec 속성으로 워크로드 및 설정에 필요한 리소스 생성

데모 애플리케이션 소개

- API Key 환경 변수와 도시 이름을 인자로 받아 날씨 출력

```
$ python weather.py seoul
{
  "success": "yes",
  "error": "",
  "city": "Seoul",
  "temperature": 16.51,
  "temperature_min": 15.66,
  "temperature_max": 16.76,
  "pressure": 1023,
  "humidity": 67,
  "feels_like": 15.97,
  "description": "clear sky"
}
```

데모 애플리케이션 컨테이너 생성

- 쿠버네티스에 배포하기 위해 실행 가능한 컨테이너 생성

```
# chanshik/weather-app:v1.0
FROM --platform=linux/arm64 python:3.12-slim

WORKDIR /app
COPY weather.py .

ENTRYPOINT ["python", "weather.py"]
```

- 컨테이너 생성 후 레지스트리에 등록

애플리케이션 리소스 정의

- 다양한 도시를 리소스로 생성해 날씨 정보 추적
- Spec: 도시와 수집에 필요한 정보 저장
- Status: 수집한 날씨와 수집 상태 등 오퍼레이터 사용 정보
- 동시에 쓰기를 시도할 때 발생하는 문제는
  쿠버네티스 플랫폼의 낙관적 동시성에 기대어 해결
- 다양한 목적으로 사용할 수 있는 저장 공간

애플리케이션 리소스 정의: Spec

- api/app/v1alpha1/weather_types.go

```
// WeatherSpec defines the desired state of Weather
type WeatherSpec struct {
    City        string `json:"city"`
    IntervalMin int    `json:"intervalMin"`
}
```

- 날씨 정보를 얻어올 도시와 주기 정보를 Spec 속성에 정의

애플리케이션 리소스 정의: Status (1)

- Status 속성에 온도, 습도 등의 날씨 데이터 정의

```
type WeatherStatus struct {
    City            string `json:"city"`
    Description     string `json:"description"`
    Temperature     string `json:"temperature"`
    TemperatureMin string `json:"temperatureMin"`
    TemperatureMax string `json:"temperatureMax"`
    Humidity        string `json:"humidity"`
    Pressure        string `json:"pressure"`
    FeelsLike       string `json:"feelsLike"`
    ...
```

애플리케이션 리소스 정의: Status (2)

- API 호출 상태와 담당한 오퍼레이터 정보를 포함하도록 정의

```
UpdateStatus    string        `json:"updateStatus"`
LastUpdatedTime *metav1.Time `json:"lastUpdatedTime,omitempty"`

// +optional
Worker string `json:"worker,omitempty"`

// +optional
WorkerAssignedTime *metav1.Time `json:"workerAssignedTime,omitempty"`
```

- 날씨 정보 수집 시각으로 LastUpdatedTime 필드 갱신

애플리케이션 리소스 정의: Status (3)

- Status 필드를 kubectl 명령 등으로 바로 볼 수 있게 정의

```
// +kubebuilder:printcolumn:name="Temperature",type=string,JSONPath=".status.temperature"
// +kubebuilder:printcolumn:name="FeelsLike",type=string,JSONPath=".status.feelsLike"
// +kubebuilder:printcolumn:name="Humidity",type=string,JSONPath=".status.humidity"
// +kubebuilder:printcolumn:name="Description",type=string,JSONPath=".status.description"
// +kubebuilder:printcolumn:name="Last Updated",type="date",JSONPath=".status.lastUpdatedTime"
// +kubebuilder:printcolumn:name="Worker",type=string,JSONPath=".status.worker"
```

```
// Weather is the Schema for the weathers API
type Weather struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec   WeatherSpec   `json:"spec,omitempty"`
    Status WeatherStatus `json:"status,omitempty"`
}
```

애플리케이션 리소스 정의: Status (4)

- kubectl get 명령을 이용해 바로 Status 속성 확인 가능

```
$ kubectl get weather
NAME           TEMPERATURE   FEELSLIKE   HUMIDITY   DESCRIPTION        LAST UPDATED
barcelona      21.28         21.03       60.00      few clouds         5s
hawaii         23.52         24.15       85.00      few clouds         2m10s
london         14.82         14.29       74.00      overcast clouds    67s
new-york-city  16.45         16.27       81.00      overcast clouds    2m10s
paris          12.38         11.81       82.00      overcast clouds    67s
seoul          16.29         15.72       67.00      clear sky          5s
```

Reconciler 구조체 정의 (1)

- Reconcile() 함수 안에서 사용할 필드 선언

```
type WeatherReconciler struct {
    client.Client
    Scheme *runtime.Scheme

    Config    *env.Config
    ClientSet kubernetes.Interface

    processingNow    bool
    processingTarget types.NamespacedName
    lock             sync.RWMutex
}
```

Reconciler 구조체 정의 (2)

- client.Client: 리소스 읽기, 쓰기 등에 사용할 클라이언트
- ClientSet: 실행 중인 컨테이너 로그를 얻어올 때 사용
- Config: 애플리케이션 컨테이너, API Key 등의 설정 정보
- processingNow: 현재 날씨 정보를 갱신하고 있다면 true
- processingTarget: 날씨 정보를 갱신하고 있는 리소스 정보
- lock: 구조체 필드를 변경할 때 사용할 lock 변수

Reconcile(): 리소스 삭제 여부 확인

- 리소스 읽을 때 NotFound 에러인 경우에는 삭제된 리소스

```
weather := &weatherv1alpha1.Weather{}
if err := r.Get(ctx, req.NamespacedName, weather); err != nil {
    if client.IgnoreNotFound(err) == nil {
        logger.Info("weather resource was deleted")
        return ctrl.Result{}, nil
    } else {
        logger.Error(err, "unable to fetch Weather")
        return ctrl.Result{}, err
    }
}
```

날씨 정보 갱신 시점 확인 (1)

- 마지막 갱신 시점에 Interval 값을 더해 현재 시각과 비교

```
func isUpdateNow(weather *weatherv1alpha1.Weather) bool {
    if weather.Status.LastUpdatedTime == nil {
        return true
    }

    interval := time.Duration(weather.Spec.IntervalMin) * time.Minute
    if weather.Status.LastUpdatedTime.Add(interval).Before(time.Now()) {
        return true
    }

    return false
}
```

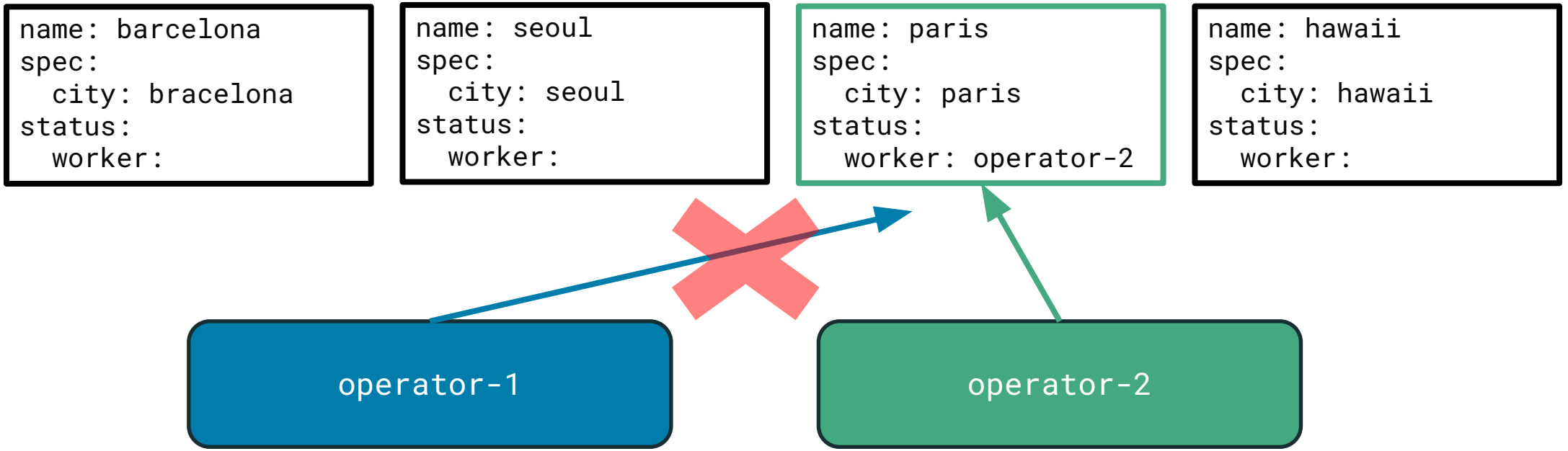
날씨 정보 갱신 시점 확인 (2)

- 갱신 시각 전이라면 갱신 시각으로 RequeueAfter 설정

```
updateNow := isUpdateNow(weather)
if updateNow == false {
    remainingTime := weather.Status.LastUpdatedTime.
        Add(time.Duration(weather.Spec.IntervalMin) * time.Minute).
        Sub(now)
    return ctrl.Result{
        Requeue:      true,
        RequeueAfter: remainingTime,
    }, nil
}
```

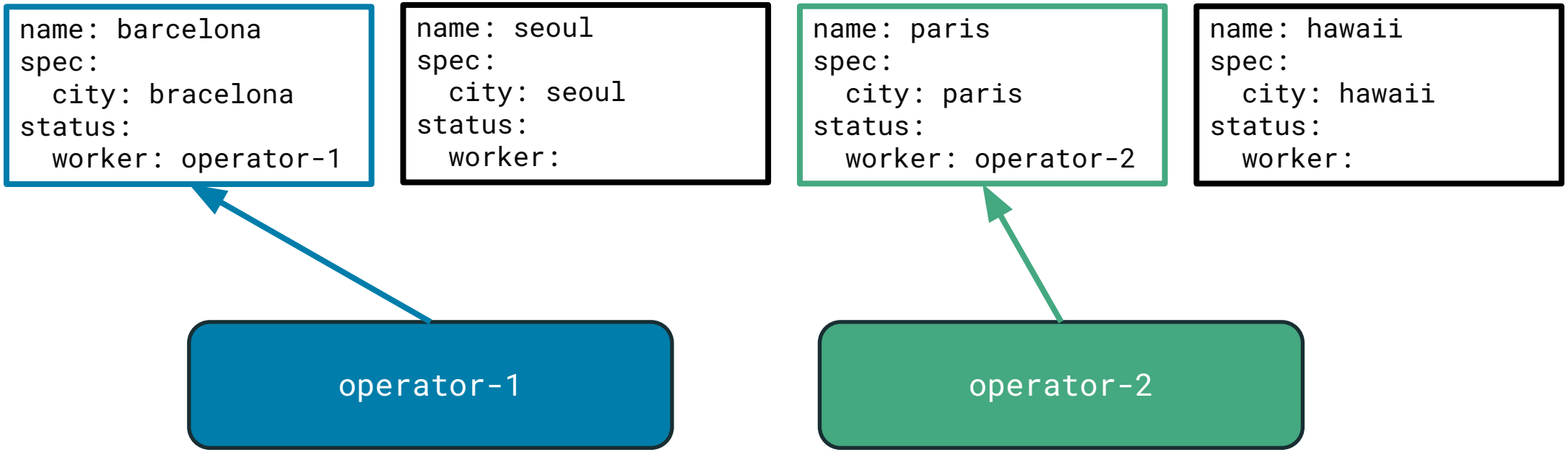
리소스 담당 Worker 할당 (1)

- Status.Worker 필드를 갱신할 때 성공한 오퍼레이터가 담당



리소스 담당 Worker 할당 (2)

- 날씨 리소스 담당 오퍼레이터 할당에 성공하면 정보 갱신



리소스 담당 Worker 할당 (3)

- Reconcile() 함수에 전달된 최신 Weather 리소스 읽기

```
func (r *WeatherReconciler) AssignWorker(
    ctx context.Context,
    namespacedName types.NamespacedName,
    workerName string,
) (bool, *weatherv1alpha1.Weather, error) {
    ...
    weather := &weatherv1alpha1.Weather{}
    err := r.Client.Get(ctx, namespacedName, weather)
    if err != nil {
        return false, nil, err
    }
```

리소스 담당 Worker 할당 (4)

- Worker 필드에 오퍼레이터 이름을 넣고 저장 시도

```
    weather.Status.Worker = workerName
    weather.Status.WorkerAssignedTime = &metav1.Time{Time: time.Now()}

    err := r.Client.Status().Update(ctx, weather)
    if err != nil {
        return false, nil, err
    }

    r.processingNow = true
    r.processingTarget = namespacedName

    return true, weather, nil
}
```

리소스 담당 Worker 할당 (5)

- 담당 오퍼레이터가 아니면 일정 시간 이후에 다시 시도
- 할당에 실패했기 때문에 기존 리소스 변경 없이 함수 종료

```
success, newWeather, err :=
    r.AssignWorker(ctx, req.NamespacedName, workerName)
if err != nil || success == false {
    logger.Info("another worker is assigned",
       "weather", weatherKey, "worker", workerName)

    return requeueResult, nil
}
```

리소스 담당 Worker 해제

- 날씨 갱신 작업이 끝나면 Worker 필드 내용 삭제

```
err := retry.RetryOnConflict(retry.DefaultRetry, func() error {
    weather := &weatherv1alpha1.Weather{}
    if err := r.Client.Get(ctx, namespacedName, weather); err != nil {
        return err
    }
    weather.Status.Worker = ""
    weather.Status.WorkerAssignedTime = nil
    return r.Client.Status().Update(ctx, weather)
})
...
```

애플리케이션 Pod 생성 (1)

- 애플리케이션을 실행할 Pod 리소스 생성

```
func buildPod(
    weather *weatherv1alpha1.Weather,
    podName, containerName,
    containerImage, apiKey string,
) *corev1.Pod {
    return &corev1.Pod{
        ObjectMeta: metav1.ObjectMeta{
            Name:      podName,
            Namespace: weather.Namespace,
        },
        ...
```

애플리케이션 Pod 생성 (2)

- Spec.City 정보를 컨테이너 인자로 전달

```
Spec: corev1.PodSpec{
    Containers: []corev1.Container{
        {
            Name: containerName, Image: containerImage,
            Command: []string{
                "python","weather.py", weather.Spec.City,
            },
            Env: []corev1.EnvVar{
                {
                    Name: "OPENWEATHERMAP_API_KEY",
                    Value: apiKey,
                },
            },
        },
```

애플리케이션 Pod 생성 (3)

- 컨트롤러 레퍼런스를 이용해 상/하위 리소스 관계 설정

```
err = ctrl.SetControllerReference(weather, weatherAppPod, r.Scheme)
if err != nil {
    logger.Error(err, "unable to set controller reference")
    r.ReleaseWorker(ctx, req.NamespacedName)
    return requeueResult, err
}

err = r.Client.Create(ctx, weatherAppPod)
if err != nil {
    logger.Error(err, "unable to create WeatherApp Pod.",
       "weather", weatherKey)
}
```

애플리케이션 Pod 생성 (4)

- Pod 생성 후 실행되어 결과를 출력할 때 까지 대기

```
weatherAppPod := &corev1.Pod{}
for {
    err := r.Get(ctx, types.NamespacedName{
                Name: podName, Namespace: weather.Namespace}, weatherAppPod)
    if err != nil {
        time.Sleep(1 * time.Second)
        continue
    }
    if weatherAppPod.Status.Phase == corev1.PodRunning {
        break
    }
    time.Sleep(1 * time.Second)
}
```

애플리케이션 실행 로그 수집 (1)

- GetLogs() 함수를 호출해 컨테이너 로그 수집

```
podLogOptions := corev1.PodLogOptions{
    Container: containerName,
}
```

```
logRequest := r.ClientSet.CoreV1().
    Pods(weather.Namespace).GetLogs(podName, &podLogOptions)

podLogs, err := logRequest.Stream(ctx)
if err != nil {
    return "", fmt.Errorf("unable to fetch logs for Pod: %v", err)
}
```

애플리케이션 실행 로그 수집 (2)

- 컨테이너 로그를 버퍼에 복사 후 문자열로 변환

```
defer func() {
    _ = podLogs.Close()
}()
buf := new(bytes.Buffer)
_, err = io.Copy(buf, podLogs)
if err != nil {
    return "", fmt.Errorf("unable to copy logs for Pod: %v", err)
}
return buf.String(), nil
```

애플리케이션 JSON 결과를 구조체로 변환 (1)

- JSON 결과 데이터를 이용해 WeatherResult 구조체 생성

```
type WeatherResult struct {
    Success         string  `json:"success"`
    Error           string  `json:"error"`
    City            string  `json:"city"`
    Description     string  `json:"description"`
    Temperature     float32 `json:"temperature"`
    TemperatureMax  float32 `json:"temperature_max"`
    TemperatureMin  float32 `json:"temperature_min"`
    Humidity        float32 `json:"humidity"`
    Pressure        float32 `json:"pressure"`
    FeelsLike       float32 `json:"feels_like"`
}
```

애플리케이션 JSON 결과를 구조체로 변환 (2)

- 문자열 JSON 결과를 구조체로 변환해 반환

```
func BuildWeatherResult(content string) (*WeatherResult, error) {
    weather := &WeatherResult{}
    if err := json.Unmarshal([]byte(content), weather); err != nil {
        return nil, err
    }

    return weather, nil
}
```

날씨 정보로 Weather.Status 필드 갱신 (1)

- 날씨 수집 시각을 Status.LastUpdatedTime 필드에 저장

```
err = retry.RetryOnConflict(retry.DefaultRetry, func() error {
    err := r.Client.Get(ctx, req.NamespacedName, weather)
    if err != nil {
        return err
    }

    if weather.Status.LastUpdatedTime == nil {
        weather.Status.LastUpdatedTime = &metav1.Time{}
    }
    weather.Status.LastUpdatedTime.Time = time.Now()
    ...
```

날씨 정보로 Weather.Status 필드 갱신 (2)

- 수집한 날씨 정보를 Status 개별 필드에 저장

```
if weatherResult.Success == "yes" {
    weather.Status.UpdateStatus = "updated"
    weather.Status.Description = weatherResult.Description
    weather.Status.City = weatherResult.City
    weather.Status.Temperature = fmt.Sprintf("%.2f", weatherResult.Temperature)
    weather.Status.FeelsLike = fmt.Sprintf("%.2f", weatherResult.FeelsLike)
    weather.Status.Pressure = fmt.Sprintf("%.2f", weatherResult.Pressure)
    weather.Status.Humidity = fmt.Sprintf("%.2f", weatherResult.Humidity)
} else {
    weather.Status.UpdateStatus = weatherResult.Error
}
return r.Client.Status().Update(ctx, weather)
```

애플리케이션 Pod 삭제 및 재호출 시점 설정

- Pod 삭제 후 다음 Interval 시점에 호출되도록 반환 값 설정

```
if err := r.Client.Delete(ctx, weatherAppPod); err != nil {
    logger.Error(err, "unable to delete Pod", "pod", podName)
    r.ReleaseWorker(ctx, req.NamespacedName)
    return requeueResult, nil
}
```

```
r.ReleaseWorker(ctx, req.NamespacedName)
return ctrl.Result{
    Requeue:      true,
    RequeueAfter: time.Duration(weather.Spec.IntervalMin) * time.Minute,
}, nil
```

오퍼레이터 실행 결과 (1)

- 오퍼레이터 Pod 두 개 배포 후 Status 정보 갱신되는 것 확인
- 동시에 두 Weather 리소스에 담당 오퍼레이터 할당

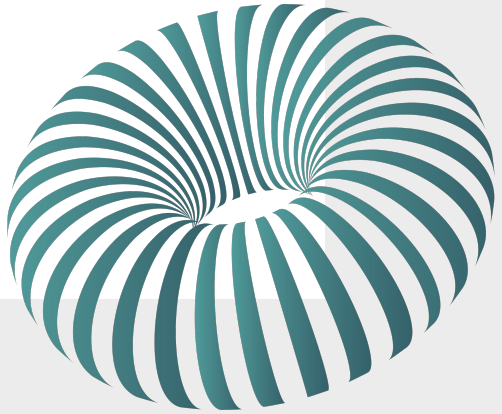
```
$ kubectl get weather
NAME            TEMPERATURE   ...   LAST UPDATED   WORKER
barcelona       23.12         ...   5m20s          weather-operator-...-gc2z7
hawaii          24.05         ...   4m39s
london          14.48         ...   4m39s
new-york-city   19.76         ...   3m58s
paris           14.00         ...   3m27s
seoul           12.74         ...   5m21s          weather-operator-...-n6g29
```

오퍼레이터 실행 결과 (2)

- 날씨 정보 갱신 후 다음 Weather 리소스 검색
- IntervalMin 속성을 고려해 갱신할 Weather 리소스 선택

```
$ kubectl get weather
NAME             TEMPERATURE   ...   LAST UPDATED   WORKER
barcelona        23.18         ...   2s
hawaii           24.05         ...   5m2s           weather-operator-...-gc2z7
london           14.48         ...   5m2s           weather-operator-...-n6g29
new-york-city    19.76         ...   4m21s
paris            14.00         ...   3m50s
seoul            12.74         ...   3s
```

정리



client-go 라이브러리

- 쿠버네티스와 상호작용하는 애플리케이션 개발에 활용
- HTTP API 방식보다 훨씬 쉽게 사용 가능
- 클러스터 내부와 외부에서 초기화 방법 모두 지원
- 리소스를 변경할 때는 낙관적 동시성을 고려해 코드 구현
- Informers 등 개발에 도움을 주는 다양한 기능 존재

kubebuilder 프레임워크

- 특별한 목적을 가진 리소스를 생성하고 관리하는 데 편리
- Spec 속성과 Status 속성을 통해 애플리케이션 상태 제어
- 복잡한 부분은 가리고 Reconcile() 함수 구현에 집중
- 처음 실행 시에 모든 리소스를 이용해 Reconcile() 함수 호출
- 주기적인 관찰이 필요하면 반환 값을 이용해 제어 가능

참고 자료

- https://kubernetes.io/docs/reference/using-api/api-concepts/
- https://kubernetes.io/docs/tasks/administer-cluster/access-cluster-api/
- https://github.com/kubernetes/client-go
- https://kubebuilder.io/introduction
- https://www.scaler.com/topics/kubernetes/kubernetes-operator/
- LINE에서 선언형 DB as a Service를 개발하며 얻은 쿠버네티스 네이티브 프로그래밍 기법 공유 https://2022.openinfradays.kr/session/3
- 쿠버네티스 커스텀 리소스 정의하고 관리하기(feat.컨트롤러) https://techblog.lycorp.co.jp/ko/define-and-manage-kubernetes-custom-resources-with-controller

Q&A

Thank you!