Kubernetes Operatori

Danijel Radaković

1 Uvod

Proširivanje Kubernetes-a

- Kubernetes omogućava upravljanje nativnih resurasa (`Pod`, `Deployment`, `CoinfigMap` itd.).
- Međutim, Kubernetes se može proširiti da upravlja novim tipovima resurasa.
- Postoje 2 načina kako se može proširiti Kubernetes da upravlja novim tipovima resurasa:
  - Korišćenjem **Custom Resource Definition** (CRD) i implementacijom operatora za definisane CRD-jeve.
  - Konfiguracijom Aggregation Layer-a.
- Mogućnost proširivanje je glavna prednost Kubernetesa u odnosu na druge orkestratore. Na ovaj način možemo registrovati svoje komponente sistema kojima će upravljati Kubernetes.
- Takođe, svoje komponente sistema možete integrasiti sa alatatima i rešenjima koji su deo Kuberenetesovog ekosistema.

Proširivanje Kubernetes-a

- Postoje popriličan broj alata u Kubernetesovom ekosistemu koji se zasnivaju na ovoj proširivosti.
- Među najpoznatijima je Crossplain, koji omogućava da pomoću Kubernetesa upravljate infrastrukturoma na različitim Cloud provajderima.
- Na primer, pomoću Crossplain alata možete upravljati EC2 instancom na AWS nalogu.
- Crossplain je zapravo IaC alat i predstavlja alterantivu za Terraform.

2 Operatori

Opšti pojmovi

- Svi resursi u Kubernetes klasteru se upravljaju od strane nekog operatora.

Resource

A resource is an endpoint in the Kubernetes API that stores a collection of **API objects** of a certain kind; for example, the built-in pods resource contains a collection of Pod objects.

API object

An entity in the Kubernetes system, represanting the part of the state of your cluster.

- Nativni resursi se upravljanju od strane operatora (`controller-manager`) koji se nalaze u `kube-system` namespace-u.

Opšti pojmovi

- Operator je u suštini `Pod/Deployment` koji sluša na izmene stanja određenih objekata, procesira ih tako da objektne dovede u željeno stanje.
- Operator se sastoji od skupa kontrolera.

Controller

In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state (doc).

- Samim tim postoje `Deployment` kontroler, `Deamonset` kontroler, `Ingress` kontroler itd. koji su deo `controller-manager`-a.

Opšti pojmovi

- Pored nativnih, postoje *custom* resursi i kontroleri.

Custom Resource

A custom resource is an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation. It represents a customization of a particular Kubernetes installation. Custom resources let you store and retrieve structured data (doc).

Custom Controller

Custom controllers can work with any kind of resource, but they are especially effective when combined with custom resources (doc).

Opšti pojmovi

Operator Pattern

The Operator pattern combines custom resources and custom controllers.

- **Primer operatora**: Hoćemo da koristimo Discord za notifikacije ali da se kreiranje servera, grupa i kanala radi preko Kubernetesa.
- Posotjala bi 3 resursa: `DiscordServer`, `DiscordGroup`, `DiscordChannel`.
- Postojla bi 3 kontrolera: `DiscordServerController`, `DiscordGroupController`, `DiscordChannelController`, koji bi bili nadležni za kreiranje, brisanje, i izmene odgovarajućih objekata.

Primeri i implementacije operatora

- Konkretne implementacije operatora:
  - Prometheus Operator,
  - MongoDB Operator,
  - CloudNativePG,
  - ostali operatori koji su deo ekosistema.
- Postoje gotovi alati koji omogućavaju implentaciju operatora (koristeći operator pattern):
  - Kubebuilder (Golang),
  - Operator SDK (Java),
  - kube-rs (Rust).
  - ostali.

3 Kubebuilder

Uvod

- U konrektnim primerimara radimo implementaciju operatora za Dojo aplikaciju. Alarmi aplikacije se prikazuju na Discord kanalu.
- Takođe, želimo da omogućimo korisnicima dinamičko kreiranje Dojo aplikacije i njenih komponenti.
- Operator ima sledeće *custom* resurse i njihove kontrolere:
  - `Dojo`: upravlja `Deployment`-om za `Dojo` aplikaciju.
  - `DiscordServer`: upravlja Discord serverom.
  - `DiscordGroup`: upravlja Discord grupom.
  - `DiscordChannel`: upravlja Discord text kanalom.
- Implementacija operatora je dostupna ovde. Postoje 4 verzije (`v1`, `v2`, `v3`, `v4`) i svaka verzija predsavlja nadogradnju prethodne.

Podešavanje okruženja

- Instalirati Go verziju `>=1.25.6`.
- Instalirati Kubebuilder verziju `>=4.11.0`.
- Instalirati k3d verziju `>=5.8.3`.
- Namesiti *bash* completion:

```
sudo sh -c 'k3d completion bash > /etc/bash_completion.d/k3d'
sudo sh -c 'kubebuilder completion bash > /etc/bash_completion.d/
kubebuilder'
```

Kreiranje klastera


```yaml
# cluster.yaml
apiVersion: k3d.io/v1alpha5
kind: Simple
metadata:
  name: local
servers: 1
```

```
k3d cluster create --config cluster.yaml
```

Kreiranje projekta

- Kreiranje Kubebulder projekta se radi pomoću sledeće komande:

```
kubebuilder init --domain jutsu.com --project-name dojo-operator --repo
github.com/DanijelRadakovic/dojo-operator
```

- `--domain`: Osnova za API Group resursa. U Kubernetesu, puni naziv grupe se formira kao `<group-name>.<domain>`. Korišćenje domena osigurava jedinstvenost unutar bilo kog klastera.
  - Primeri naziva grupa: `core.jutsu.com`, `billing.jutsu.com`.
- `--repo`: Naziv Go modula.

Kreiranje custom resursa (proširavanje API-a)

- Projekat je inicijalizovan i samim tim nema definisan ni jedan *custom* resurs i kontroler, pa ih je neopdhodno dodati.
- Želimo da definišemo `Dojo` *custom* resurs i omogućimo upravljanje preko Kubernetesa.
- Da bi to postigli treba da:
  - kreiramo Custom Resource Definition (CRD) za `Dojo` resurs,
  - kontoler koji će upravljati `Dojo` objektima.

Custom Resource Definition (CRD)

- Za definisanje custom resursa koristi se Custom Resource Definition (CRD) resurs.
- To je resurs koji kreira *RESTful resource path* i OpenAPI v3.0 šemu na API Serveru. Šema se koristi za validaciju REST zahteva.
- Samim tim mnoge funkcionalnosti koje podržava OpenAPI v3.0 podržava i CRD uz neke razlike i ograničenja (doc).

Custom Resource Definition (CRD)

- Svaki CRD mora da ima sledeće:
  - `ApiVersion`: Grupa i verzija kojoj resurs pripada.
  - `Kind`: Naziv resursa napisan u PascalCase formatu (npr. `Dojo`).
  - `Scope`: Određuje da li je resurs vezan za namespace ili je na nivou celog klastera. Vrednosti su: `Namespaced` ili `Cluster`.
  - `Spec`: Željeno stanje resursa (ono što korisnik definiše u YAML-u).
  - `Status`: Trenutno stanje resursa (ono što Operator upisuje nakon posmatranja klastera).
  - `Singular`: Jednina naziva resursa, koristi se u `kubectl` komandama (npr. `dojo`).
  - `Plural`: Množina naziva resursa, koristi se u URL putanjama API-ja i `kubectl` listama (npr. dojos).

Kreiranje custom resursa (proširavanje API-a)

- Kubebuilder nam omogućava da na jednostavan način kreiramo CRD i kontroler.

```
kubebuilder create api --group core --version v1 --kind Dojo --resource --
controller --plural dojos
```

- `--group`: Naziv grupe.
- `--version`: Verzija resursa. Konvencije i način upravljanje verzijama je definisano ovde.
- `--kind`: Naziv resursa. Pun naziv resursa je `dojo.core.jutsu.com`.
- `--resource`: Generiše kod za CRD.
- `--controller`: Generiše kod za kontroler.
- `--pluar`: Množina naziva resursa.

Kreiranje custom resursa (proširavanje API-a)

- Prethodna komanda izgeneriše sledeće fajlove:

```
INFO api/v1/dojo_types.go
INFO api/v1/groupversion_info.go
INFO internal/controller/suite_test.go
INFO internal/controller/dojo_controller.go
INFO internal/controller/dojo_controller_test.go
```

- Od posebnog interesa su nam sledeći fajlovi:
  - `api/v1/dojo_types.go` - koristi se za konfiguraciju CRD.
  - `internal/controller/dojo_controller.go` - koristi se za implementaciju kontrolera.
  - `internal/controler/dojo_controller_test.go` - koristi se za pisanje *unit* testova za kontroler.

Implementacija CRD-a

- Što se tiče obaveznih elemenata CRD-a, Kubebuilder je već dosta stvari uradio za nas: `ApiVersion`, `Kind`, `Scope`, `Singular`, `Plural`.
- Od nas se očekuje da definišemo `Spec` i `Status` resursa, s obzirom na to da su specifični za svaki resurs.
- Oni se definišu pomoću `DojoSpec` i `DojoStatus` struktura koje se definisane u `api/v1/dojo_types.go` fajlu.

Implementacija CRD-a


```
// DojoSpec defines the desired state of Dojo
type DojoSpec struct {
  // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
  // Important: Run "make" to regenerate code after modifying this file
  // The following markers will use OpenAPI v3 schema to validate the
value
  // More info: https://book.kubebuilder.io/reference/markers/crd-
validation.html

  // foo is an example field of Dojo. Edit dojo_types.go to remove/update
  // +optional
  Foo *string `json:"foo,omitempty"`
}
```

Implementacija CRD-a

- Za definisanje željenog stanja resursa koristi se `Spec` resursa. Treba da izgleda ovako:

```yaml
apiVersion: core.jutsu.com/v1
kind: Dojo
metadata:
  name: tokyo-jujutsu-high
spec:
  accountId: Masamichi Yaga # Naziv korisnika koji kreira aplikaciju.
  title: Tokyo Jujutsu High # Naziv Dojo-a.
  replicas: 3 # Broj replikaca Dojo aplikacije.
  database: Postgres # Gde se čuvaju podaci: Postgres ili Mongo.
  credentialsRef: # Naziv Secret objekta koji sadrži kredencijale baze.
   name: dojo
   namespace: default
```

Implementacija CRD-a (DojoSpec)


```
// DojoSpec defines the desired state of Dojo
type DojoSpec struct {
  // AccountId which owns the Dojo application.
  AccountId string `json:"accountId"`
  // Title of the Dojo application.
  Title string `json:"title"`
  // Database type for the application.
  // +kubebuilder:default:="Postgres"
  Database *Database `json:"database"`
  // CredentialsRef is a reference to Secret which contains the database
credentials.
  // Postgres: Reference the Secret which points to the `owner` of
cnpg.Database.
  // Mongo: Not yet supported.
```

Implementacija CRD-a (DojoSpec) (ii)


```
  CredentialsRef corev1.SecretReference `json:"credentialsRef"`
  // Replicas is number of application instances to run.
  // +kubebuilder:validation:Minimum=0
  // +kubebuilder:default:=1
  // +optional
  Replicas *int32 `json:"replicas"`
}
```

- S obzriom na to da su ovo input vrednosti, neophodno ih je validirati.
- Validaciju nam omogućavaju Kubebuilder markups.
- Obratiti pažnju da li je neki atribut strukture pokazivač ili ne.
- Kada koristiti `int` a kada `*int`, ili `string` i `*string`?

Implementacija CRD-a (DojoSpec) (iii)

- Ukoliko je atribut obavezan onda treba koristiti vrednosti a ne pokazivače: `int`, `stirng`.
- Ukoliko je atribut opcioni onda koristiti pokazivače `*int` `*string`.
- Ovim načinom izbegavate nejasnoće u vašem kodu. Na primer, ako je neki atribut opcioni i tipa je `string`, ukoliko se ne prosledi vrednost, biće `""`. Ne znamo da li je vrednost nedostajuća ili je sa razlogom postavljena ta vrednost. Takođe, može da izazove neželjenje efekte kod API Servera kada se radi Server Side Apply.

Implementacija CRD-a (Enumeracije)


```
// +kubebuilder:validation:Enum=Postgres;Mongo
type Database string
const (
  DatabasePostgres Database = "Postgres"
  DatabaseMongo    Database = "Mongo"
)
```

```
// DojoSpec defines the desired state of Dojo
type DojoSpec struct {
  // Database type for the application.
  // +kubebuilder:default:="Postgres"
  Database *Database `json:"database"`
}
```

Implementacija CRD-a (DojoStatus)


```
// DojoStatus defines the observed state of Dojo.
type DojoStatus struct {
  // Standard condition types include:
  // - "Available": the resource is fully functional
  // - "Progressing": the resource is being created or updated
  // - "Degraded": the resource failed to reach or maintain its desired
state
  //
  // The status of each condition is one of True, False, or Unknown.
  // +listType=map
  // +listMapKey=type
  // +optional
  Conditions []metav1.Condition `json:"conditions,omitempty"`
}
```

Implementacija CRD-a (DojoStatus)

- `Status` resursa prestavlja trenutno stanje resursa.
- Kontroler je nadležan da upravlja ovom strukturom.
- Možete staviti sve što vam je neophodno da lakške pratite stanje objekta.
- Preporuke kako pisati `Status` resursa je dostupno ovde.
- Jedna od generalni preporuka jeste da se koristi `Conditions` sa barem 3 elemenata:
  - Available: resurs je spreman za korićenje.
  - Progressing: resurs se procesira tako do dostigne novo željeno stanje.
  - Degraded: resurs prilikom neke greške nije use da dostigne željeno stanje.
- S obzirom na to da smo omogućili u `Spec` resursa broj replika, u `Status` resursa moramo da pratimo koliko je replika dostupno a na koliko replika se čega da budu dostupne (indentično statusima `Deployment` resursa).

Implementacija CRD-a (DojoStatus)


```
// DojoStatus defines the observed state of Dojo.
type DojoStatus struct {
  // Conditions represent the current state of the Dojo resource.
  // +listType=map
  // +listMapKey=type
  // +optional
  Conditions []metav1.Condition `json:"conditions,omitempty"`
  // Credentials represents the Secret that is created for database
credentials
  // +optional
  Credentials corev1.SecretReference `json:"credentials,omitempty"`
  // ReadyReplicas is the number of Pods created by the Deployment that
have the Ready condition.
  // +optional
```

Implementacija CRD-a (DojoStatus) (ii)


```
  ReadyReplicas int32 `json:"readyReplicas,omitempty"`
  // UpdatedReplicas is the number of Pods created by the Deployment that
are running the most recent version of the Pod template.
  // +optional
  UpdatedReplicas int32 `json:"updatedReplicas,omitempty"`
  // AvailableReplicas is the number of Pods created by the Deployment
that have been ready for at least minReadySeconds.
  // +optional
  AvailableReplicas int32 `json:"availableReplicas,omitempty"`
  // ReadyStatus is a human-readable string representing the ratio of
ready replicas to desired replicas (e.g., "1/3").
  // +optional
  ReadyStatus string `json:"readyStatus,omitempty"`
}
```

Implementacija CRD-a (DojoStatus)

- Svi atributi moraju da su opcioni jer moraju biti prazni prilikom prvog definisanja objekata. Kontoler kasnije upravlja statusom objekta.
- **Napomena**: Status resursa je deo API-a i mora se voditi računa o **backward** i **forward** komptatibilnošću.
- Drugi controleri (ili alati) mogu da koriste status objekat da bi pratili njegovo stanje i znali da odreguju u određenim situacijama.
- Ukoliko se napravi **brekable** izmena, ostali kontroleri i alati neće raditi dobro.
- Na primer ArgoCD alat koristi `Progressing` status da prati da li se objekat i dalje procesira od strane kontrolera.

Implementacija CRD-a

- Definisali smo `Spec` i `Status` resursa i spremni smo na osnovu toga da izgenerišemo CRD.
- Generianje CRD-a:

```
make manifest generate
```

- Izgenerisan CRD: `config/crd/bases/core.jutsu.com_dojos.yaml`

Implementacija kontolera

- Implementacija kontrolera se nalazi u `internal/controller/dojo_controller.go`.
- Kao što je već napomenuto, kontoler je petlja koja čega na izmene koje su se primenile na `Dojo` objekte i procesira ih u skaldu sa nekom logikom.
- Ta petlja se naziva **reconcile** petlja a struktura koja implementira petju se naziva **Reconciler**.

```
// DojoReconciler reconciles a Dojo object
type DojoReconciler struct {
  client.Client
  Scheme *runtime.Scheme
}
```

Implementacija kontolera


```
// (ommited kubebuilder markups for RBAC)

// Reconcile is part of the main kubernetes reconciliation loop which aims
to
// move the current state of the cluster closer to the desired state.
// For more details, check Reconcile and its Result here:
// - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.23.0/pkg/
reconcile
func (r *DojoReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
  _ = logf.FromContext(ctx)
  // TODO(user): your logic here
  return ctrl.Result{}, nil
}
```

Implementacija kontolera


```
func (r *DojoReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
  dojo := &corev1.Dojo{}
  if err := r.Get(ctx, req.NamespacedName, dojo); err != nil {
    return ctrl.Result{}, client.IgnoreNotFound(err)
  }
  fmt.Println("Name: ", dojo.Name)
  fmt.Println("Title:", dojo.Spec.Title)
  fmt.Println("Owner:", dojo.Spec.AccountId)
  fmt.Println("Replicas:", *dojo.Spec.Replicas)
  fmt.Println("CredentialsRef:", dojo.Spec.CredentialsRef)

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

Implementacija kontolera

- `Reconcile` metoda ima jedan parameter (zanemajurući `context`) `ctrl.Request`. `ctrl.Request` sadrži `name` i `namespace` `Dojo` objekta nad kojim se desila izmena i treba da se procesira od strane Reconcilera.
- Pomoću `name` i `namepsace` (i `Kind`) možemo jedinstveno da identifikujemo bilo koji objekat unutar clustera. Zbog toga se često u samoj biblioteci NamespacedName provlači u drugim strukturama.
- Na onsovu `name` i `namespace` moramo da dobavimo kompletan objekat iz klastera. Kad je objekat dobavljen, treba ga procesirati na osnovu `Spec` i `Status` vrednosti objekta.

Implementacija kontolera (v1)

- Dobavljanje se radi pomoću `r.Get(ctx, req.NamespacedName, &corev1.Dojo)`.
- Ukoliko nije pronađen to znači da se desilo brisanje objekta i nema potrebe sa dodatnim procesirajem. Time završavamo *reconcile* petlju.
- Za uspešno završavanje *reconcile* petlje neophodno je vratiti `ctrl.Result{}, nil`.
- Za neuspešno izvršavanje, bitno je vratiti `error` koji nije `nil`.
- Zbog toga, u slučaju brisanja objekata koristimo pomoćnu metodu `client.IgnoreNotFound(err)`.
- Takođe, Reconciler u pozadini koristi `client.Client` pa se njegove metode mogu koristi u *reconcile* petlji.

Instalacija CRD i pokretanje operatora

- Imamo jednostavnu implementaciju kontolera, samim tim smo spremni da instaliramo CRD i pokrenemo operator u lokalu.
- Instalacija CRD-a:

```
make intall
```

```
$ kubectl get crds
NAME                                  CREATED AT
...
clusters.postgresql.cnpg.io           2026-02-05T00:19:40Z
databases.postgresql.cnpg.io          2026-02-05T00:19:40Z
dojos.core.jutsu.com                  2026-02-24T01:16:59Z
...
```

Instalacija CRD i pokretanje operatora

- Pokretanje operatora:

```
make run
```

```
INFO    setup   starting manager
INFO    starting server {"name": "health probe", "addr": "[::]:8081"}
INFO    Starting EventSource    {"controller": "dojo", "controllerGroup":
"core.jutsu.com", "controllerKind": "Dojo", "source": "kind source:
*v1.Dojo"}
INFO    Starting Controller     {"controller": "dojo", "controllerGroup":
"core.jutsu.com", "controllerKind": "Dojo"}
INFO    Starting workers        {"controller": "dojo", "controllerGroup":
"core.jutsu.com", "controllerKind": "Dojo", "worker count": 1}
```

Instalacija CRD i pokretanje operatora

- Instalacijom CRD-a proširili smo API Server.
- Na API Serveru je kreiran novi RESTful API endpoint:

```
/apis/core.jutsu.com/v1/namespaces/*/dojos/...
```

- Samim tim možemo preko `kubectl` alata da korstimo resurs.

Instalacija CRD i pokretanje operatora


```
cat <<EOF | kubectl apply -f -
apiVersion: core.jutsu.com/v1
kind: Dojo
metadata:
  name: tokyo-jujutsu-high
spec:
  accountId: Masamichi Yaga
  title: Tokyo Jujutsu High
  replicas: 3
  database: Postgres
  credentialsRef:
    name: dojo
    namespace: default
EOF
```

Instalacija CRD i pokretanje operatora


```
$ kubect get dojos
NAME                 AGE
tokyo-jujutsu-high   3m18s
```

```
INFO    Starting workers        {"controller": "dojo", "controllerGroup":
"core.jutsu.com", "controllerKind": "Dojo", "worker count": 1}
Name:  tokyo-jujutsu-high
Title: Tokyo Jujutsu High
Owner: Masamichi Yaga
Replicas: 3
CredentialsRef: {dojo default}
```

Instalacija CRD i pokretanje operatora

- Primere za testiranje operatora možemo definisati u `config/samples/core_v1_dojo.yaml`.
- Primeniti ih na cluster pomoću: `kubectl apply -f config/samples/core_v1_dojo.yaml`.
- Ili koristiti kustomize: `kubectl apply -k config/samples`.

4 Dojo operator v2

Implementacija operatora

- Ustanovili smo da se *reconcile* petlja okida na promene `Dojo` objekata.
- Sada želimo da unapredimo petlju tako da kreiramo Deployment za Dojo aplikaciju sa brojem replika definisanim u `Spec` objekta.
- Važno je napomenuti da je Reconciler **stateless**. Ne pamti stanja iz prethodnog izvršavanja.
- S obzirom na to da se događaji mogu duplicirati, važno je da Reconciler bude **idempotentan**.

Idempotency

Idempotency is the property where an operation can be applied multiple times without changing the result beyond the initial application.

Implementacija operatora

- U slučaju da se ne ispoštuje idempotentnost, može doći do beskonačne petlje.
- Način kojim se Reconciler pravi idempotetnim jeste korišćenjem `Status` objekta.
- Na osnovu `Status` objekta Reconciler zna u kojoj fazi procesiranja objekta se nalazi i kako dalje da nastavi sa procesiranjem objekta.
- Zbog toga je bitno dobro modelovati `Status` resursa.
- Bitno je napomenuti da se ceo Kubernetes bazira na **level-based** dizajnu umesto **edge-based**.

Implementacija operatora

Level-based design

The system must operate correctly given the desired state and the current/observed state, regardless of how many intermediate state updates may have been missed. Edge-triggered behavior must be just an optimization (doc).

- Na primer, izršavamo *reconcile* petlju i prilikom izvršavanja se dese 5 promena objekta (npr. korisnik je pomoću `kubectl` 5 puta izmenilo objekat). Ne interesuje nas prethodne 4 promene, intereseuje nas samo poslednja.
- Postoje 3 povratne vrednosti kojima se vraca rezultat u `Reconcile` metodi:
  -
- Pogledati Reconciler implementaciju.

Implementacija operatora

- Prilikom implentacije neophodno je identifikovati greške od kojih se Reconciler ne može oporativit.
- U skladu sa tim greškama `Status` objekta treba izmeniti na odgovarajući način. Odnosno podesiti `Reason` i `Status(true/false)` vrednosti u odgovarajućim `Conditions`.
- Na primer, od `409 Conflict` greške se može oporaviti jer naglašava da ne radimo sa najnovijom verzijom objekta, i treba opet okinuti petlju koja će raditi sa najnovijom verzijom.
- Međutim, od greške koja nastaje da se Deployment ne može kreirati jer je nešto pogrešno konfigurisano (selector, ownership) je greška on koje se Reconciler ne može opraviti.

Implementacija operatora

- U našoj implementaciji postoje 3 `Conditions`: `Available`, `Progressing`, `Degraded`.
- `Available` je `true` ako postoji sa barem jedna replika Dojo aplikacije, u suprotnom je `false`.
- `Degraded` je `true` ukoliko je nastala greška od koje nema oporavka, u suprotnom je `false`.
- `Progressing` je `true` ukoliko Reconciler procesira objekat u željeno stranje.
- `Degraded` i `Progressing` su međusovno isključivi, odnosno ukoliko je `Degraded=True`, onda mora da je `Progressing=False` i obrnuto.
- Zbog toga postoje metode `setUnrecoverableErrorStatus` koja postaljva `Degraded=True,Progressing=False`, i njena inverzna metoda `setProgressStatus`.

Implementacija operatora

- `Available` nije međusobno isključiv sa `Degraded` i `Progressing`:
  - `Available=True,Progressing=True`: Postoje 3 replike, a Reconciler radi na tome da ih skalira na 5 jer je tako definisano u `Spec` objekta.
  - `Available=True,Degraded=True`: Postoje 3 replike, a žejeno stanje je 5 replika. U međuvremenu se desila greška od koje nema oporavka.
- Reconciler ne sme da menja `Spec` objekta. Sme da menja samo `Status` (kasnije ćemo videti `Scale` i `Finalizer`).

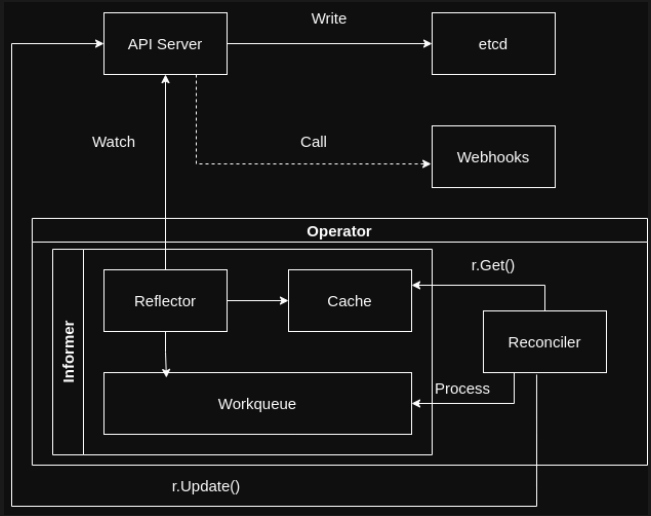
Implementacija operatora

- U petlji ne treba da se nalazi logika koja čega resurs da se kreira (npr. EC2 istanca, Discord kanal).
- Svaka petlja ima `timeout` koji kada istekne prekine isršavanje te iteracije.
- Takođe ostali zahtevi se ne mogu procesirati dok se iteracija te petlje na završi (ukoliko niste podeseili `concurency`).
- Pametnije bi bilo da završite sa iteracijom petlje i okinete petlju ponovo nakog određenog vremana sa `ctrl.Request{After: 30 * Seconds}`.

Implementacija operatora

- **Zlatno pravilo**: U jednoj iteraciji petlje može se raditi samo jedna izmena objekta.
- API Server ima evidenciju trenutne verzije svakog objekta. Verzija objekta se nalazi u `metadata.resourceVersion`.
- Svaka izmena objekta (bilo `Spec` ili `Status`) povećava verziju objekta.
- Ukoliko se na API Server pošalje objekat (pomoću `r.Update()`) koji nema najnoviju verziju, API Server vraća `409 Confilct` grešku.
- Zbog toga je dobra praksa da se posle svakog `r.Update()`, uradi provera `!apierros.IsConflic(err)` kako bi ignorisali tu grešku i opet okinuli petlju.
- Sasvim je normalno da se *reconcile* petlja više puta okine kako bi se došlo do željeneog stanja.

Implementacija operatora



Slika 1: Komunikacija između komponenti

Implementacija operatora

- API Server je komponenta kojoj se šalju REST zahtevi za izmene objekata.
- API Server je jedina komponenta koja sme da upisuje u *etcd* bazu.
- Pre upisivanja objekta u bazu pozivaju se *Webhook*-ovi.
- Kada se desi neka promena na API Serveru, API Server radi broadcast tako da sve ostale komponente koje su zaintersovane za taj događaj mogu da odreaguju.

Implementacija operatora

- Operator koristi `ctrl.Client` koji u pozadini pokreće *Informer* komponentu.
- Informer obuvata sledeće komponente:
  - `Reflector`: Zadužen je da gleda izmene objekata koje su se desile na API Serveru a od interesa su za operator.
  - `Cache`: Čuva kompletne objekte (`Metadata`, `Spec`, `Status`) koji su dobavljeni od strane Reflector-a.
  - `Workqueue`: Metapodaci objekata (`namespace/name`) dobavjeni od strane Reflector-a informer stavlja u Workqueue. Workqueue okida reconciler petlju.
- **Napomena**: Ovo je pojednostavljen dizajn Informera i prava implementacija je dosta složenija.

Implementacija operatora

- Za *Workqueue* `namespace/name` predstavljaju ključeve i *Workqueue* je zadužen da **uklanja duplikate ključeva**.
- Ukoliko se jedan objakat promenio 5 puta imaće isti ključ u *Workqueue* i samim tim će uraditi uklanjanje duplikata i biti samo jedan element. Ovaj princip poštuje **level-based** dizajn po kojem nas interesuje samo poslednja izmena, prethodne 4 se ignorišu.
- `r.Get()` dobavlja objekte iz keša. Sve komponente Kubernetesa imaju keš kako ne bi opteretili API Server sa zahtevima.
- `r.Update()` direktno komunicira sa API Server. Ovaj zahtev API Server obrađuje i radi broadcast koji bi u nekom trenutno trebao da se propagira do operatora i opet okine *reconcile* petlju.

Implementacija operatora

- S obzrim na to da API Server radi broadcast može se deisti da keš nema poslednju verziju objekta prilikom izvršavanja *reconcile* petlje i da dođe do `409 Conflict` greške ukoliko se u toj iteraciji petlje radi `r.Update()`.
- Postoji način da se objekat direktno dobavi sa API Servera: `r.APIReader`. Međutim, ovo način komunikacije ne treba uzlopotrebljavati kako ne bi udarili u *rate limit* API Servera.
- Rate limit se može konfigurisati prilikom kreiranja klastera pomoću `k3d`: `--max-requests-inflight, Default: 400` (doc)

Implementacija operatora


```
func (r *DojoReconciler) deploymentForDojo(dojo *corev1.Dojo)
(*appsv1.Deployment, error) {
  image := "danijelradakovic/dojo:0.1.0-alpine"
  dojoLabels := r.labels(dojo.Name)
  dep := &appsv1.Deployment{...}

  // Set the ownerRef for the Deployment
  if err := ctrl.SetControllerReference(dojo, dep, r.Scheme); err != nil {
    return nil, err
  }
  return dep, nil
}
```

Implementacija operatora

- Podesili smo da je Dojo ima *ownership* nad Deployment objektom.
- Ovo nam omogućava da prilikom brisanja Dojo objekta, Kubernetes *Carbage Collector* će obrisati Deployment za nas.
- Postavljanje ownership-a nad objektom podrazumeva dodeljivanje `metadata.ownerReferences` objektu.

Implementacija operatora


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
  name: tokyo-jujutsu-high
  namespace: default
  ownerReferences:
  - apiVersion: core.jutsu.com/v1
    blockOwnerDeletion: true
    controller: true
    kind: Dojo
    name: tokyo-jujutsu-high
    uid: b9fd0c5d-c0bd-4206-8bca-8e18e85423ad
```

Implementacija operatora

- S obzirom na to da smo postavili *ownership* Deployment objektu, moramo da podesimo kontroler da posmatra izmena za taj deployment.
- Ne posmatramo sve Deployment objekte nego samo one koji imaju *ownership* od strane Dojo objekta (optimizujemo keš i memoriju operatora).

```
// SetupWithManager sets up the controller with the Manager.
func (r *DojoReconciler) SetupWithManager(mgr ctrl.Manager) error {
  return ctrl.NewControllerManagedBy(mgr).
    For(&corev1.Dojo{}).
    Owns(&appsv1.Deployment{}).
    Named("dojo").
    Complete(r)
}
```

Implementacija operatora

- Na ovaj način ako neko izmeni ili obriše Deployment objekat, okinuće se *reconcile* petlja i konfigurisati Deployment u skladu sa `Spec` Dojo objekta.

```
$ kubectl get pods,deployment,dojo
NAME                                      READY   STATUS    RESTARTS   AGE
pod/tokyo-jujutsu-high-79669f6d67-5zn56   1/1     Running   0          60m
pod/tokyo-jujutsu-high-79669f6d67-7kxsd   1/1     Running   0          60m
pod/tokyo-jujutsu-high-79669f6d67-g2htr   1/1     Running   0          60m

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/tokyo-jujutsu-high    3/3     3            3           60m

NAME                                      AGE
dojo.core.jutsu.com/tokyo-jujutsu-high    60m
```

Implementacija operatora


```
$ kubectl scale deployment tokyo-jujutsu-high --replicas 1
$ kubectl get pods,deployment,dojo
NAME                                      READY  STATUS       RESTARTS AGE
pod/tokyo-jujutsu-high-79669f6d67-4hwz5   1/1    Running      0        16s
pod/tokyo-jujutsu-high-79669f6d67-5zn56   1/1    Terminating  0        63m
pod/tokyo-jujutsu-high-79669f6d67-7kxsd   1/1    Running      0        63m
pod/tokyo-jujutsu-high-79669f6d67-g2htr   1/1    Terminating  0        63m
pod/tokyo-jujutsu-high-79669f6d67-g64p8   1/1    Running      0        16s

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/tokyo-jujutsu-high    3/3     3            3           63m

NAME                                     AGE
dojo.core.jutsu.com/tokyo-jujutsu-high   63m
```

5 Dojo operator v3

Implementacija operatora

- Deployment resurs može da se scalira koristeći `kubctl scale` komandu. Želimo da omogućimo isto podršu za naš Dojo resurs
- Takođe, *HorizontalPodAutoscaler* (HPA) skalira na identičan način pa omogućujemo i njegovu integraciju.
- Skaliranje se zapravo dešava na `/scale` podresursu.

Implementacija operatora

- Neophodno je dodati `Selector` polje u statusu zbog HPA.

```
// DojoStatus defines the observed state of Dojo.
type DojoStatus struct {
  ...
  // Selector is required for HPA to work with CRDs.
  // It must be the string representation of the label selector.
  // +optional
  Selector string `json:"selector,omitempty"`
}
```

Implementacija operatora

- Konfigurisati *markup* sa `/scale` podresurs tako da bira obuhvata `readyReplicas` i `selector` iz `Status` resursa.

```
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status
// +kubebuilder:resource:path=dojos
// +kubebuilder:subresource:scale:specpath=.spec.replicas,
//statuspath=.status.readyReplicas,selectorpath=.status.selector
// Dojo is the Schema for the dojos API
type Dojo struct {
  metav1.TypeMeta `json:",inline"`
  ...
}
```

Implementacija operatora

- Želimo da nam ispisuje dodatna polja Dojo objekta kada koristimo `kubect get dojo`.

```
// +kubebuilder:resource:shortName=dj,categories={all}
// +:printcolumn:name="READY",type="string",JSONPath=".status.readyStatus"
// +:printcolumn:name="UP-TO-DATE",type="integer",JSONPath=".status.upd"
// +:printcolumn:name="AVAILABLE",type="integer",JSONPath=".status."
// +:printcolumn:name="ACCOUNT",type="string",JSONPath=".spec.accountId"
// +:printcolumn:name="STORAGE",type="string",JSONPath=".spec.storage"
// +:printcolumn:name="AGE",type="date",JSONPath=".metadata.creati"
type Dojo struct {...}
```

Implementacija operatora

- Da bi nam grupe radile moramo da obrišemo i instaliramo CRD ponovo.

```
make uninstall
make manifest generate install run
kubect apply -k config/sample
```

Implementacija operatora


```
$ kubect  get all

NAME                                        READY   STATUS  RESTARTS AGE
pod/tokyo-jujutsu-high-79669f6d67-jvf5r   1/1     Running 0        104s
pod/tokyo-jujutsu-high-79669f6d67-nhjm6   1/1     Running 0        104s
pod/tokyo-jujutsu-high-79669f6d67-sfjds   1/1     Running 0        104s
...

NAME                                      READY   UP-TO-DATE   AVAILABLE
ACCOUNT          AGE
dojo.core.jutsu.com/tokyo-jujutsu-high   3/3     3            3
The Higher-Ups   104s
```

Implementacija operatora


```
$ kubectl scale dojo tokyo-jujutsu-high --replicas 2
$ kubectl get all
NAME                                        READY STATUS      RESTARTS AGE
pod/tokyo-jujutsu-high-79669f6d67-jvf5r   1/1   Terminating 0        6m33s
pod/tokyo-jujutsu-high-79669f6d67-nhjm6   1/1   Running     0        6m33s
pod/tokyo-jujutsu-high-79669f6d67-sfjds   1/1   Running     0        6m33s
...

NAME                                      READY   UP-TO-DATE   AVAILABLE
ACCOUNT          AGE
dojo.core.jutsu.com/tokyo-jujutsu-high   2/2     2            2
The Higher-Ups   6m33s
```

Upravljanje `reconcile` petljom

Exponential Backoff Requeue

Subresursi za /status i /scale

6 Dojo operator v4

Kreiranje Postgres baze

- Dojo aplikacija zahteva konekciju ka bazi i da se krenedicijali nalaze u `env var` koji će se popuniti iz `Secret` objekta.
- **Problem**: kako znati koji `Secret` korisiti i šta treba da bude njegov sadržaj?
- Jedno rešenje bi bilo da kreiramo *custom* resurs `Postgres` i kontroler koji bi kreirao bazu kao `Statefulset` i `Secret` sa odgovarjućim kredencijalima.
- Međutim, ova implementacija bi bila minimalna i pitanje koliko mi moglo da posluži u produkcionim okruženjima.
- Bolja opcija je da koristimo već gotove operatore za upravljanje Postgres bazom, kao što je CloudNativePG.

Kreiranje Postgres baze

- Resurse koje `CloudNativePG` operator nudi se nalaze ovde.
- Od posebnog značaja su Bootstrap i Database resurs koji omogućava kreiranje baze unutar klustera i `Secret` objekta sa odgovarajućim kredencijalima.
- Instalacija CloudNativePG operatora:

```
helm repo add cnpg https://cloudnative-pg.github.io/charts
helm upgrade --install cnpg \
  --namespace cnpg-system \
  --create-namespace \
  cnpg/cloudnative-pg
```

Kreiranje Postgres baze


```
cat <<EOF | kubectl apply -f -
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: dojo
spec:
  instances: 1
  bootstrap:
    initdb:
      database: dojo
      owner: dojo
      postInitApplicationSQL:
        - CREATE TABLE weapons(id text not null, name text not null,
PRIMARY KEY(id))
```

Kreiranje Postgres baze (ii)


```
        - ALTER TABLE weapons OWNER TO dojo;
  storage:
    size: 1Gi
EOF
```

Kreiranje Postgres baze


```
$ kubectl get cluster,secrets
NAME                              AGE   INSTANCES   READY   STATUS
PRIMARY
cluster.postgresql.cnpg.io/dojo   48s   1           1       Cluster in
healthy state   dojo-1

NAME                       TYPE                              DATA   AGE
secret/dojo-app            kubernetes.io/basic-auth          11     48s
secret/dojo-ca             Opaque                            2      48s
secret/dojo-replication    kubernetes.io/tls                 2      48s
secret/dojo-server         kubernetes.io/tls                 2      48s
```

Kreiranje Postgres baze


```
$ kubectl exec -it dojo-1 -- bash
postgres@dojo-1:/$ psql
psql (18.1 (Debian 18.1-1.pgdg13+2))
Type "help" for help.

postgres=# \l

   Name    |  Owner   | Encoding | Locale Provider | Collate | Ctype |
-----------+----------+----------+-----------------+---------+-------+
 dojo      | dojo     | UTF8     | libc            | C       | C     |
 postgres  | postgres | UTF8     | libc            | C       | C     |
 template0 | postgres | UTF8     | libc            | C       | C     |
           |          |          |                 |         |       |
 template1 | postgres | UTF8     | libc            | C       | C     |
```

Kreiranje Postgres baze


```
$ kubectl get -o yaml secrets dojo-app
apiVersion: v1
data:
  dbname: ZG9qbw==
  fqdn-jdbc-uri: amRiYzpwb3N0Z3Jlc3FsOi8vZG9qby1ydy5kZWZhdWx0LnN2Yy5...
  fqdn-uri: cG9zdGdyZXNxbDovL2Rvam86WmJBdWJydTM0ZmRqeGJVQTlTSThSaXlZ...
  host: ZG9qby1ydw==
  jdbc-uri: amRiYzpwb3N0Z3Jlc3FsOi8vZG9qby1ydy5kZWZhdWx0OjU0MzIvZG9q...
  password: WmJBdWJydTM0ZmRqeGJVQTlTSThSaXlZS1d1MGRiSjRoS0FJeVhRdmN3...
  pgpass: ZG9qby1ydzo1NDMyOmRvam86ZG9qbzpaYkF1YnJ1MzRmZGp4YlVBOVNJOF...
  port: NTQzMg==
  uri: cG9zdGdyZXNxbDovL2Rvam86WmJBdWJydTM0ZmRqeGJVQTlTSThSaXlZS1d1M...
  user: ZG9qbw==
  username: ZG9qbw==
```

Implementacija operatora

- Pogledati Reconlicer implementaciju.

```
kubectl apply -k config/sample
# change the pod name accordingly
kubectl exec -it tokyo-jujutsu-high-7fd46f69bf-8c797 -- ash

$ wget -qO- --post-data="" "http://localhost:8080/weapon?id=0&weapon=
katana"
$ wget -qO- --post-data="" "http://localhost:8080/weapon?id=1&weapon=
ninjaStar"
$ wget -qO- --post-data="" "http://localhost:8080/weapon?id=2&weapon=
ninjaSword"
$ wget -qO-  "http://localhost:8080/weapon"
```

Implementacija operatora


```
// SetupWithManager sets up the controller with the Manager.
func (r *DojoReconciler) SetupWithManager(mgr ctrl.Manager) error {
  return ctrl.NewControllerManagedBy(mgr).
    For(&corev1.Dojo{}).
    Owns(&appsv1.Deployment{}).
    Owns(&k8scorev1.Secret{}).
    Watches(
      &k8scorev1.Secret{},
      handler.EnqueueRequestsFromMapFunc(r.findDojosForSecret),
      builder.WithPredicates(predicate.LabelChangedPredicate{}),
    ).
    Named("dojo").
    Complete(r)
}
```

Implementacija operatora

- `Watch()` sekcija naznačava da kontroler gleda sve Secret objekte unutar klaster i čuva ih u keš Informera. Pre upisivanja u *Workqueue* secreti se filtriraraju na osnovu `findDojosForSecret`. Metoda će vratiti kolekciju `namespace/name` Dojo objekata koji ce se biti prosleđeni u *Workqueue*.
- Na ovaj način smo omogućili da svi Dojo objekti koji su zainteresovani za kredencijale baze biti procesirani kada nastane promene vezane za kredencijale.

Dodatna pojašnjenja

Pisanje testova