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 prošititi 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 Kubernetes-a 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 Kuberenetes-ovog ekosistema.

Proširivanje Kubernetes-a

- Postoje popriličan broj alata u Kubernetes-ovom ekosistemu koji se zasnivaju na ovoj proširivosti.
- Među najpoznatijima je Crossplain, koji omogućava da pomoću Kubernetes-a 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 Kubernetes-a.
- 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 kanal.
- 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.

Podešavanje okruženja

- Instalirati Go, verzije `>=1.25.6`.
- Instalirati Kubebuilder, verzije `>=4.11.0`.
- Instalirati k3d, verzije `>=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 u 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 tip resursa koji kreira RESTful resource path i OpenAPI v3.0 šemu na API Serveru koju 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.

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 sa nas: `ApiVersion`, `Kind`, `Scope`, `Singular`, `Plural`.
- Od nas se očekuje da definišemo `Spec` i `Status` podresurse, s obzirom na to da su specifični za svaki resurs pojedinačno.
- 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

- `Spec` podresurom se definiše željeno stanje resursa. Recimo da 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, ukoliko je neko atribut opicioni 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 kreiranje 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 na 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 nalaze 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 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 (v1)


```
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
}
```

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`.

Implementacija operatora (v2)

- 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 se Reconciler implementira da 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 (v2)

- U slučaju da se ne ispoštuje idempotentnost, može doći do beskonačne petlje.
- Način kojim se Reconciler pravi idempotetnim jeste korišćenjem statusa objekta.
- Na osnovu statusa 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 (v2)

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).

- zlatno pravilo

Upravljanje `reconcile` petljom

- ne sme da se radi wait() zato sto reconcile petlja ima timeout, drugi requestovi iz working queue ne mogu da se obradjuju. Dovoljno sam blokirajuce akcje blokiraju, samo jos ovo treba
- S obzirom da smo definisali validaciju u CRD, ne treba opet da radimo validaciju u reconcile petrlji.

Ima queue promena koje su se desile, i cim posotji queue treba da se napravi da bude idempotent.

U queue se cuva namespace/name i radi deduplication kako bi smanjio frekveciju promena. Na primer u jednom loop-u, promenio se shop1/shop resource 10 puta, umesto da u queue bude 10 elemenata postojace samo jedan jer ce se duplikati ukloniti. Nacin kako da se napravi da reconciler bude idempotent jeste da bude da se koristi status.

Ne bi trebao da controller dira Spec. Samo status. Subresursi za /status i /scale

Upravljanje `reconcile` petljom (ii)

Da bi smo na pravilan način implenetirali `reconcile` petlju, moramo prvo bolje da znamo kako API Server i ostali mehanizmi funkcionišu u pozadini.

Pisanje testova

Pisanje testova

Kreiranje Postgres baze

- Aplikacija zahteva konekciju ka bazi. Očekuje 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 je `Database` resurs koji omogućava kreiranje baze unutar klustera i `Secret` objekta sa odgovarajućim kredencijalima.
- Sve što naj je ostalo jeste da proširimo `Spec` sekciju u kojoj je moguće definisati tip baze koji se koristi i naziv `Secret` objekta.