RAPORT 11

DEPARTAMENT: OPTOSPINTRONICA

Gavrila Vlad-Theodor

***Kubernetes.CRD.Operatori***

**Definitie:**

* Kubernetes este un sistem open-source de orchestrare a containerelor pentru automatizarea implementării, scalarii și gestionării software-ului. Proiectat inițial de Google, proiectul este acum întreținut de Cloud Native Computing Foundation.
* Kubernetes funcționează cu diferite durate de rulare a containerelor, cum ar fi containerd și CRI-O. Adecvarea sa pentru rularea și gestionarea sarcinilor mari de lucru native din cloud a condus la adoptarea pe scară largă a acestuia în centrul de date. Există mai multe distribuții ale acestei platforme – de la furnizori independenți de software (ISV), precum și oferte găzduite pe cloud de la toți furnizorii majori de cloud public.

**Arhitectura:**

A diagram of a software server

Description automatically generated

* Kubernetes definește un set de blocuri de construcție („primitive”) care oferă în mod colectiv mecanisme care implementează, întrețin și scala aplicații bazate pe CPU, memorie sau valori personalizate. Kubernetes este cuplat și extensibil pentru a răspunde nevoilor diferitelor sarcini de lucru. Componentele interne, precum și extensiile și containerele care rulează pe Kubernetes se bazează pe API-ul Kubernetes. Platforma își exercită controlul asupra resurselor de calcul și stocare prin definirea resurselor ca obiecte, care pot fi apoi gestionate ca atare.
* Kubernetes urmează arhitectura primară/replica. Componentele Kubernetes pot fi împărțite în cele care gestionează un nod individual și cele care fac parte din planul de control.

**Planul de control:**

Nodul principal Kubernetes se ocupă de planul de control Kubernetes al clusterului, gestionând volumul de lucru al acestuia și direcționând comunicarea în sistem. Planul de control Kubernetes constă din diverse componente, fiecare proces propriu, care pot rula atât pe un singur nod master, cât și pe mai multe master care acceptă clustere de înaltă disponibilitate. Diferitele componente ale planului de control Kubernetes sunt după cum urmează.

**etcd:**

etcd este un magazin de date persistent, ușor, distribuit, cheie-valoare (dezvoltat inițial pentru Container Linux). Stochează în mod fiabil datele de configurare ale clusterului, reprezentând starea generală a clusterului în orice moment dat. etcd favorizează consistența față de disponibilitate în cazul unei partiții de rețea (vezi teorema CAP). Consecvența este crucială pentru programarea și operarea corectă a serviciilor.

**Server API:**

* Serverul API servește API-ul Kubernetes folosind JSON prin HTTP, care oferă atât interfața internă, cât și externă pentru Kubernetes. Serverul API procesează, validează cererile REST și actualizează starea obiectelor API în etcd, permițând astfel clienților să configureze încărcături de lucru și containere în nodurile de lucru. Serverul API folosește API-ul de urmărire al etcd pentru a monitoriza clusterul, pentru a lansa modificări critice de configurare sau pentru a restabili orice divergențe ale stării clusterului la starea dorită, așa cum este declarată în etcd.
* Ca exemplu, un operator uman poate specifica că trebuie să ruleze trei instanțe ale unui anumit „pod” (vezi mai jos), iar etcd stochează acest fapt. Dacă controlerul de implementare constată că rulează doar două instanțe (în conflict cu declarația etcd), programează crearea unei instanțe suplimentare a acelui pod.

**Planificator:**

* Planificatorul este o componentă extensibilă care selectează nodul pe care îl rulează un pod neprogramat (unitatea de bază a workload-urilor care urmează să fie programate), pe baza disponibilității resurselor și a altor constrângeri. Planificatorul urmărește alocarea resurselor pe fiecare nod pentru a se asigura că volumul de lucru nu este programat peste resursele disponibile. În acest scop, planificatorul trebuie să cunoască cerințele de resurse, disponibilitatea resurselor și alte constrângeri sau directive de politică furnizate de utilizator, cum ar fi calitatea serviciului, cerințele de afinitate/anti-afinitate și localitatea datelor. Rolul planificatorului este de a potrivi „oferta” de resurse cu „cererea” sarcinii de lucru.
* Kubernetes permite rularea mai multor programatoare într-un singur cluster. Ca atare, plug-in-urile de planificare pot fi dezvoltate și instalate ca extensii în proces pentru planificatorul nativ vanilla, rulându-l ca un planificator separat, atâta timp cât sunt conforme cu cadrul de planificare Kubernetes. Acest lucru le permite administratorilor de cluster să extindă sau să modifice comportamentul planificatorului Kubernetes implicit în funcție de nevoile lor.

**Controlorii:**

* Un controler este o buclă de reconciliere care conduce starea actuală a clusterului către starea dorită, comunicând cu serverul API pentru a crea, actualiza și șterge resursele pe care le gestionează (de exemplu, pod-uri sau puncte finale de serviciu).
* Un exemplu de controler este un controler ReplicaSet, care se ocupă de replicare și scalare prin rularea unui număr specificat de copii ale unui pod în cluster. Controlerul se ocupă, de asemenea, de crearea podurilor de înlocuire dacă nodul de bază eșuează. Alte controlere care fac parte din sistemul de bază Kubernetes includ un controler DaemonSet pentru a rula exact un pod pe fiecare mașină (sau un subset de mașini) și un controler de job pentru rularea pod-urilor care rulează până la finalizare (de exemplu, ca parte a unui job batch). Selectoarele de etichete fac adesea parte din definiția controlerului care specifică setul de pod-uri pe care le gestionează un controler.
* Managerul de controler este un singur proces care gestionează mai multe controlere Kubernetes de bază (inclusiv exemplele descrise mai sus) și este distribuit ca parte a instalării standard Kubernetes.
* Controlere personalizate pot fi, de asemenea, instalate în cluster, permițând în continuare extinderea comportamentului și API-ului Kubernetes atunci când sunt utilizate împreună cu resurse personalizate (vezi mai jos resurse personalizate, controlere și operatori).

**Noduri:**

Un nod, cunoscut și sub numele de lucrător sau minion, este o mașină în care sunt desfășurate containere (încărcături de lucru). Fiecare nod din cluster trebuie să ruleze un timp de rulare a containerului, precum și componentele menționate mai jos, pentru comunicarea cu configurația de rețea primară a acestor containere.

**Kubelet:**

Kubelet este responsabil pentru starea de funcționare a fiecărui nod, asigurându-se că toate containerele de pe nod sunt sănătoase. Se ocupă de pornirea, oprirea și întreținerea containerelor de aplicații organizate în pod-uri conform instrucțiunilor planului de control. kubelet monitorizează starea unui pod, iar dacă nu este în starea dorită, pod-ul se reinstalează în același nod. Starea nodului este transmisă la fiecare câteva secunde prin mesaje inimii către serverul API. Odată ce planul de control detectează o defecțiune a nodului, este de așteptat ca un controler de nivel superior să observe această schimbare de stare și să lanseze pod-uri pe un alt nod sănătos.

**Durata de rulare a containerului:**

* Un timp de rulare a containerului este responsabil pentru ciclul de viață al containerelor, inclusiv lansarea, reconcilierea și uciderea containerelor.
* kubelet interacționează cu timpii de rulare a containerului prin intermediul Container Runtime Interface (CRI), care decuplă întreținerea Kubernetes-ului de bază de implementarea reală a CRI.
* Inițial, kubelet a interfațat exclusiv cu runtime-ul Docker printr-un „dockershim”. Cu toate acestea, din noiembrie 2020 până în aprilie 2022, Kubernetes a depreciat shim-ul în favoarea interfeței directe cu containerul prin containerd sau înlocuirea Docker cu un timp de execuție care este compatibil cu Container Runtime Interface (CRI). Odată cu lansarea v1.24 în mai 2022, „dockershim” a fost eliminat în întregime.
* Exemple de durate de rulare a containerelor populare care sunt compatibile cu kubelet includ containerd (acceptat inițial prin Docker), rkt și CRI-O.

**kube-proxy:**

kube-proxy este o implementare a unui proxy de rețea și a unui echilibrator de încărcare și acceptă abstracția serviciului împreună cu celelalte operațiuni de rețea. Acesta este responsabil pentru rutarea traficului către containerul corespunzător, pe baza IP-ului și a numărului de port al cererii primite.

**Namespace:**

În Kubernetes, namespace-urile sunt utilizate pentru a separa resursele pe care le gestionează în colecții distincte și care nu se intersectează. Acestea sunt destinate utilizării în medii cu mulți utilizatori repartizați în mai multe echipe sau proiecte sau chiar medii separate, cum ar fi dezvoltarea, testarea și producția.

**Pods:**

* Unitatea de planificare de bază din Kubernetes este un pod, care constă dintr-unul sau mai multe containere care sunt garantate a fi amplasate pe același nod. Fiecărui pod din Kubernetes i se atribuie o adresă IP unică în cluster, permițând aplicațiilor să utilizeze porturi fără riscul de conflict. În interiorul podului, toate containerele se pot referi unul la altul.
* Un container se află în interiorul unui pod. Containerul este cel mai de jos nivel al unui micro-serviciu, care deține aplicația care rulează, bibliotecile și dependențele acestora.
* Un volum, cum ar fi un director de disc local sau un disc de rețea, poate fi definit într-un pod și făcut accesibil containerelor care locuiesc în acesta. Pod-urile pot fi gestionate manual prin API-ul Kubernetes sau gestionarea lor poate fi delegată unui controler. Astfel de volume sunt, de asemenea, baza pentru caracteristicile Kubernetes ale ConfigMaps (pentru a oferi acces la configurație prin sistemul de fișiere vizibil pentru container) și Secrete (pentru a oferi acces la acreditările necesare pentru a accesa în siguranță resursele de la distanță, prin furnizarea acelor acreditări în sistemul de fișiere, vizibile numai pentru containerele autorizate).

**Sarcini de lucru:**

Kubernetes acceptă mai multe abstracții ale sarcinilor de lucru care sunt la un nivel superior față de pod-uri simple. Acest lucru permite utilizatorilor să definească și să gestioneze în mod declarativ aceste abstracții la nivel înalt, în loc să fie nevoiți să gestioneze singuri podurile individuale. Câteva dintre aceste abstracții, susținute de o instalare standard a Kubernetes, sunt descrise mai jos.

**ReplicaSets, ReplicationControllers și Deployments(implementari):**

* Scopul unui ReplicaSet este de a menține un set stabil de replică care rulează la un moment dat. Ca atare, este adesea folosit pentru a garanta disponibilitatea unui număr specificat de Pod-uri identice. Se poate spune că ReplicaSet este un mecanism de grupare care îi permite lui Kubernetes să mențină numărul de instanțe care au fost declarate pentru un pod dat. Definiția unui ReplicaSet folosește un selector, a cărui evaluare va avea ca rezultat identificarea tuturor podurilor care sunt asociate cu acesta.
* Un ReplicationController, similar unui ReplicaSet, servește aceluiași scop și se comportă similar cu un ReplicaSet, care este de a se asigura că va exista întotdeauna un număr specificat de replici pod. Volumul de lucru ReplicationController a fost predecesorul unui ReplicaSet, dar în cele din urmă a fost depreciat în favoarea ReplicaSet pentru a utiliza selectoare de etichete bazate pe set.
* **"Deployments"** în Kubernetes joacă un rol esențial în managementul și automatizarea desfășurării (deployment) aplicațiilor în cadrul unui cluster Kubernetes. Iată detalii despre rolul și funcționalitățile principale ale deployments în Kubernetes:
* **Gestionarea Stării Dorite (Desired State)**: Deployments permit definirea stării dorite pentru aplicații. Aceasta include numărul de replici ale aplicației care trebuie să ruleze și specificația containerului (imaginea, variabilele de mediu, volumele etc.). Kubernetes lucrează continuu pentru a se asigura că starea actuală a clusterului corespunde stării dorite definite în deployment.
* **Scalare**: Deployments permit scalarea automată sau manuală a numărului de replici ale unei aplicații. Asta înseamnă că poți mări sau micșora numărul de instanțe ale aplicației în funcție de necesități.
* **Actualizări și Rollback**: Kubernetes facilitează actualizările aplicațiilor prin modificarea specificației containerului în deployment. Acest lucru poate include schimbarea versiunii imaginii containerului. Kubernetes se ocupă de actualizarea treptată a containerelor, minimizând timpul de nefuncționare. În cazul unei actualizări eșuate, Kubernetes poate efectua un rollback la o versiune anterioară.
* **Auto-reparare**: Kubernetes monitorizează și asigură sănătatea aplicațiilor. Dacă un container eșuează, este înlocuit automat, asigurând astfel disponibilitatea și fiabilitatea aplicației.
* **Declarativitate**: Configurația pentru deployments este declarativă, ceea ce înseamnă că specifici "ce" să se întâmple (starea dorită), nu "cum" să se întâmple. Kubernetes se ocupă de "cum", simplificând astfel managementul aplicațiilor.
* **Strategii de Deployment**: Kubernetes suportă diferite strategii de deployment, cum ar fi "RollingUpdate" (actualizare treptată) și "Recreate" (recrearea tuturor instanțelor simultan). Aceste strategii permit controlul modului în care se desfășoară actualizările.
* **Managementul Resurselor**: Deployments pot include și specificații pentru alocarea resurselor (cum ar fi CPU și memorie) și politici pentru restart, asigurând eficiența utilizării resurselor în cluster.
* **Izolare și Securitate**: Prin utilizarea namespaces și a altor mecanisme de securitate, deployments pot fi izolate pentru a limita accesul și a crește securitatea aplicațiilor.

**StatefulSets:**

* StatefulSets sunt controlere care impun proprietățile unicității și ordonării între instanțe ale unui pod și pot fi utilizate pentru a rula aplicații cu stare. În timp ce scalarea aplicațiilor fără stat este doar o chestiune de adăugare a mai multor poduri care rulează, a face acest lucru pentru sarcinile de lucru cu stare este mai dificilă, deoarece starea trebuie păstrată dacă un pod este repornit. Dacă aplicația este mărită sau redusă, poate fi necesar să fie redistribuită starea.
* Bazele de date sunt un exemplu de sarcini de lucru cu stare. Când sunt rulate în modul de înaltă disponibilitate, multe baze de date vin cu noțiunea de instanță primară și instanțe secundare. În acest caz, noțiunea de ordonare a instanțelor este importantă. Alte aplicații precum Apache Kafka distribuie datele între brokerii lor; prin urmare, un broker nu este același cu altul. În acest caz, noțiunea de unicitate a instanței este importantă.

**DaemonSets:**

DaemonSets sunt responsabili pentru a se asigura că un pod este creat pe fiecare nod din cluster. În general, majoritatea încărcărilor de lucru se scalează ca răspuns la un număr de replici dorit, în funcție de disponibilitatea și cerințele de performanță, după cum este nevoie de aplicație. Cu toate acestea, în alte scenarii, poate fi necesară implementarea unui pod în fiecare nod din cluster, mărind numărul total de poduri pe măsură ce sunt adăugate noduri și colectând gunoiul pe măsură ce sunt eliminate. Acest lucru este util în special pentru cazurile de utilizare în care volumul de lucru are o anumită dependență de nodul real sau de mașina gazdă, cum ar fi colecția de jurnal, controlerele de intrare și serviciile de stocare.

**Servicii:**

* Un serviciu Kubernetes este un set de pod-uri care funcționează împreună, cum ar fi un nivel al unei aplicații cu mai multe niveluri. Setul de pod-uri care constituie un serviciu sunt definite de un selector de etichete. Kubernetes oferă două moduri de descoperire a serviciilor, folosind variabile de mediu sau folosind DNS Kubernetes. Descoperirea serviciului atribuie serviciului o adresă IP stabilă și un nume DNS, iar sarcina echilibrează traficul într-un mod round-robin la conexiunile de rețea ale acelei adrese IP dintre podurile care se potrivesc cu selectorul (chiar dacă eșecurile determină transferul pod-urilor de la o mașină la alta ). În mod implicit, un serviciu este expus în interiorul unui cluster (de exemplu, podurile back-end pot fi grupate într-un serviciu, cu solicitările de la pod-urile front-end echilibrate între ele), dar un serviciu poate fi expus și în afara unui cluster (de ex., pentru ca clienții să ajungă la podurile front-end).

**Volumele:**

Sistemele de fișiere din containerul Kubernetes oferă stocare efemeră, în mod implicit. Aceasta înseamnă că o repornire a podului va șterge orice date de pe astfel de containere și, prin urmare, această formă de stocare este destul de limitantă în orice altceva decât în ​​aplicații banale. Un volum Kubernetes oferă stocare persistentă care există pe durata de viață a podului în sine. Această stocare poate fi folosită și ca spațiu pe disc partajat pentru containerele din pod. Volumele sunt montate în anumite puncte de montare din container, care sunt definite de configurația podului și nu se pot monta pe alte volume sau nu se pot conecta la alte volume. Același volum poate fi montat în diferite puncte din arborele sistemului de fișiere prin containere diferite.

**ConfigMaps și Secrete:**

* O provocare obișnuită a aplicației este să decidă unde să stocheze și să gestioneze informațiile de configurare, dintre care unele pot conține date sensibile. Datele de configurare pot fi ceva la fel de bine definite ca proprietățile individuale sau informații cu granulație grosieră, cum ar fi fișiere de configurare întregi, cum ar fi documentele JSON sau XML. Kubernetes oferă două mecanisme strâns legate pentru a face față acestei nevoi, cunoscute sub denumirea de ConfigMaps și Secrets, ambele permitând efectuarea modificărilor de configurare fără a necesita o reconstrucție a aplicației.
* Datele din ConfigMaps și Secrets vor fi disponibile pentru fiecare instanță a aplicației la care aceste obiecte au fost legate prin implementare. Un Secret și/sau un ConfigMap este trimis la un nod numai dacă un pod de pe acel nod o cere, care va fi stocat doar în memorie pe nod. Odată ce pod-ul care depinde de Secret sau ConfigMap este șters, copia din memorie a tuturor Secretelor și ConfigMaps legate este ștearsă, de asemenea.
* Datele dintr-un ConfigMap sau Secret sunt accesibile podului prin unul dintre următoarele moduri:
* Ca variabile de mediu, care vor fi consumate de kubelet din ConfigMap atunci când containerul este lansat;
* Montat într-un volum accesibil în sistemul de fișiere al containerului, care acceptă reîncărcarea automată fără a reporni containerul.
* Cea mai mare diferență dintre un Secret și un ConfigMap este că Secretele sunt concepute special pentru a conține date sigure și confidențiale, deși nu sunt criptate în mod implicit și necesită o configurare suplimentară pentru a securiza pe deplin utilizarea Secretelor în cadrul clusterului. Secretele sunt adesea folosite pentru a stoca date confidențiale sau sensibile, cum ar fi certificate, acreditări pentru a lucra cu registre de imagini, parole și chei ssh.

**Etichete și selectoare:**

* Kubernetes le permite clienților (utilizatori sau componente interne) să atașeze chei numite etichete la orice obiect API din sistem, cum ar fi pod-uri și noduri. În mod corespunzător, selectoarele de etichete sunt interogări împotriva etichetelor care se rezolvă la obiecte care se potrivesc. Când este definit un serviciu, se pot defini selectoarele de etichete care vor fi utilizate de ruterul de servicii/echilibratorul de încărcare pentru a selecta instanțele pod către care va fi direcționat traficul. Astfel, simpla schimbare a etichetelor pod-urilor sau schimbarea selectoarelor de etichete pe serviciu poate fi folosită pentru a controla ce poduri primesc trafic și care nu, care pot fi folosite pentru a susține diverse modele de implementare, cum ar fi implementări albastru-verde sau A/B testarea. Această capacitate de a controla dinamic modul în care serviciile utilizează resursele de implementare oferă o cuplare slabă în cadrul infrastructurii.
* De exemplu, dacă pod-urile unei aplicații au etichete pentru un nivel de sistem (cu valori precum frontend, backend, de exemplu) și un release\_track (cu valori precum canary, production, de exemplu), atunci o operație pe toate backend și canary nodurile pot folosi un selector de etichete, cum ar fi:

tier=backend AND release\_track=canary

* În exemplul dat:
* Se presupune că pod-urile unei aplicații au două tipuri de etichete: una pentru nivelul sistemului (tier) și alta pentru traseul de lansare (release\_track).
* Etichetele tier pot avea valori cum ar fi frontend, backend, etc., care indică rolul sau funcția acelui pod în cadrul aplicației.
* Etichetele release\_track pot avea valori precum canary, production, etc., care indică stadiul ciclului de lansare în care se află pod-ul (de exemplu, testare, producție).
* Când se dorește să se efectueze o operație pe toate pod-urile care sunt atât în nivelul backend, cât și în traseul de lansare canary, se poate utiliza un selector de etichete care combină aceste două criterii. Selectorul de etichete ar fi exprimat ca secventa de cod de mai sus.
* Acest selector indică faptul că doar resursele care au eticheta tier cu valoarea backend și în același timp eticheta release\_track cu valoarea canary vor fi selectate. Aceasta este o modalitate puternică de a organiza și de a interacționa cu resursele dintr-un cluster Kubernetes, permițând gestionarea complexă și flexibilă a diferitelor componente ale aplicației. Selectorii de etichete pot fi utilizați în diverse contexte, cum ar fi în politici de rețea, în seturi de replici, în servicii Kubernetes, și în alte resurse pentru a defini cum interacționează și cum sunt gestionate aceste resurse.
* La fel ca etichetele, selectoarele de câmpuri permit selectarea resurselor Kubernetes. Spre deosebire de etichete, selecția se bazează pe valorile atributelor inerente resursei selectate, mai degrabă decât pe categorizarea definită de utilizator. metadata.name și metadata.namespace sunt selectori de câmp care vor fi prezenți pe toate obiectele Kubernetes. Alți selectori care pot fi utilizați depind de tipul obiectului/resursei.

**Add-ons(Adaugari):**

Adaugarile funcționează la fel ca orice altă aplicație care rulează în cluster: sunt implementate prin pod-uri și servicii și sunt diferite doar prin faptul că implementează caracteristici ale clusterului Kubernetes. Pod-urile pot fi gestionate de Deployments, ReplicationControllers și așa mai departe. Există multe tipuri de adaugari, iar lista crește. Unele dintre cele mai importante sunt:

**DNS:**

DNS (Domain Name System) în Kubernetes joacă un rol crucial în localizarea serviciilor și a podurilor din cadrul unui cluster. Iată cum funcționează și care sunt caracteristicile principale ale DNS-ului în Kubernetes:

1. **Descoperirea Serviciilor**: Kubernetes DNS este utilizat pentru descoperirea serviciilor în cadrul unui cluster. Fiecare serviciu Kubernetes este automat asignat cu un nume DNS. Când o aplicație dintr-un pod dorește să comunice cu un alt serviciu, poate folosi acest nume DNS, ceea ce simplifică procesul de descoperire și comunicare între servicii.
2. **DNS Integrat**: Kubernetes vine cu o soluție DNS integrată (de obicei CoreDNS), care este configurată automat pentru a servi cererile DNS din cadrul clusterului. CoreDNS este un server DNS flexibil și extensibil, care poate fi configurat pentru a servi cereri DNS specifice pentru Kubernetes.
3. **Nume DNS pentru Servicii**: Numele DNS ale serviciilor Kubernetes sunt formate după un model standard. De exemplu, un serviciu numit my-service în namespace-ul my-namespace va avea un nume DNS de forma my-service.my-namespace.svc.cluster.local, unde svc.cluster.local este sufixul standard pentru serviciile Kubernetes.
4. **DNS pentru Pod-uri**: Fiecare pod are de asemenea un nume DNS bazat pe adresa sa IP și pe namespace. Aceasta permite comunicarea directă între pod-uri, dacă este necesar.
5. **Rezolvarea Numei Serviciilor Externe**: CoreDNS în Kubernetes poate fi configurat pentru a redirecționa cererile pentru anumite domenii către servere DNS specifice. Acest lucru este util pentru rezolvarea numelor serviciilor externe clusterului.
6. **Configurarea Custom**: Deși Kubernetes oferă un DNS pre-configurat, administratorii de cluster pot personaliza setările DNS pentru a se potrivi nevoilor specifice ale aplicațiilor lor, inclusiv adăugarea de politici și reguli personalizate.
7. **Rolul în Înaltă Disponibilitate**: DNS-ul în Kubernetes contribuie la înalta disponibilitate a serviciilor prin furnizarea unui mecanism consistent și fiabil pentru localizarea serviciilor și a podurilor, indiferent de modificările infrastructurii de rețea.
8. **Securitate**: CoreDNS în Kubernetes poate fi configurat cu politici de securitate pentru a controla accesul la servicii și pentru a proteja comunicațiile în cadrul clusterului.

**Web UI:**

Aceasta este o interfață de utilizare web de uz general pentru clusterele Kubernetes. Permite utilizatorilor să evalueze performanta aplicatiilor in cluster-ul Kubernetes.

**Monitorizarea resurselor containerelor:**

Oferirea unui timp de rulare fiabil al aplicației și posibilitatea de a o scala în sus sau în jos ca răspuns la sarcinile de lucru înseamnă a putea monitoriza în mod continuu și eficient performanța sarcinii de lucru. Monitorizarea resurselor containerelor oferă această capacitate prin înregistrarea valorilor despre containere într-o bază de date centrală și oferă o interfață de utilizare pentru răsfoirea datelor respective. cAdvisor-ul este o componentă a unui nod slave care oferă o capacitate limitată de monitorizare a metrică. Există, de asemenea, conducte complete de metrice, cum ar fi Prometheus, care pot satisface majoritatea nevoilor de monitorizare.

**Monitorizarea costurilor containerelor:**

Aplicațiile de monitorizare a costurilor Kubernetes permit defalcarea costurilor pe poduri, noduri, spații de nume și etichete. Trei valori esențiale de urmărit sunt cheltuielile zilnice în cloud, costul pe CPU furnizat și solicitat, alocarea istorică a costurilor.

**Înregistrare la nivel de cluster:**

Jurnalele ar trebui să aibă o stocare și un ciclu de viață separat, independent de noduri, poduri sau containere. În caz contrar, defecțiunile nodurilor sau ale podului pot cauza pierderea datelor despre evenimente. Capacitatea de a face acest lucru se numește înregistrare la nivel de cluster, iar astfel de mecanisme sunt responsabile pentru salvarea jurnalelor de container într-un magazin central de jurnal cu interfață de căutare/navigare. Kubernetes nu oferă stocare nativă pentru datele de jurnal, dar se pot integra multe soluții de logare existente în clusterul Kubernetes.

**Spatiile de stocare PV/PVC:**

Spatiile de stocare în Kubernetes, cunoscute sub numele de Volume-uri Persistente (Persistent Volumes - PV) și Reclamații de Volume Persistente (Persistent Volume Claims - PVC), sunt concepte esențiale pentru managementul stocării de date într-un cluster Kubernetes. Iată o descriere detaliată a acestor două componente:

1. **Volume Persistente (PV)**: Un PV este o resursă în clusterul Kubernetes care reprezintă o unitate de stocare. PV-urile sunt independente de ciclul de viață al podurilor individuale și oferă o modalitate de a rezerva resurse de stocare în cluster. PV-urile pot proveni din diverse surse de stocare, cum ar fi discuri locale, NFS, iSCSI, cloud storage (de exemplu, AWS EBS, Azure Disk, Google Persistent Disk), și altele.
2. **Reclamații de Volume Persistente (PVC)**: Un PVC este o cerere de stocare de către un utilizator. PVC-urile permit utilizatorilor să solicite resurse de stocare specifică, cum ar fi capacitatea și modul de acces (de exemplu, ReadWriteOnce, ReadOnlyMany, ReadWriteMany). PVC-urile sunt asociate cu PV-urile prin procesul de binding, unde Kubernetes caută un PV disponibil care să corespundă cerințelor PVC-ului și îl asociază cu acesta.
3. **Procesul de Binding**: Atunci când un PVC este creat, Kubernetes caută un PV disponibil care corespunde cerințelor specificate în PVC (capacitate, mod de acces, clase de stocare, etc.). Dacă se găsește un PV compatibil, este "legat" de PVC. Odată legat, PV-ul este exclusiv pentru PVC și nu poate fi folosit de alte PVC-uri.
4. **Clase de Stocare (Storage Classes)**: Kubernetes permite definirea de "clase de stocare" care specifică diferite tipuri și politici de stocare. Clasele de stocare pot fi folosite pentru a automatiza provizionarea de noi PV-uri bazate pe cerințele PVC-urilor. Fiecare clasă de stocare poate avea propriile parametri și politici, cum ar fi tipul backend-ului de stocare, politici de snapshot, etc.
5. **Provizionare Dinamică și Statică**: Kubernetes suportă atât provizionarea statică (unde administratorii de cluster creează PV-uri manual), cât și provizionarea dinamică (unde noi PV-uri sunt create automat atunci când sunt necesare, pe baza unei clase de stocare).
6. **Ciclul de Viață**: PV-urile au un ciclu de viață independent de podurile care le utilizează. Un PV poate fi "reclamat" (recycled), "păstrat" (retained) sau "șters" (deleted) după ce PVC-ul asociat este șters, în funcție de politica de reutilizare configurată.
7. **Acces Multiplu**: PV-urile pot suporta diferite moduri de acces, cum ar fi accesul în mod single-writer (ReadWriteOnce) sau multi-reader/multi-writer (ReadWriteMany sau ReadOnlyMany), în funcție de tipul de stocare și de capabilitățile acesteia.

**Depozitare:**

* Containerele au apărut ca o modalitate de a face software-ul portabil. Containerul conține toate pachetele de care aveți nevoie pentru a rula un serviciu. Sistemul de fișiere furnizat face containerele extrem de portabile și ușor de utilizat în dezvoltare. Un container poate fi mutat de la dezvoltare la testare sau producție fără modificări de configurare sau relativ puține.
* Din punct de vedere istoric, Kubernetes a fost potrivit doar pentru servicii fără stare. Cu toate acestea, multe aplicații au o bază de date, care necesită persistență, ceea ce duce la crearea de stocare persistentă pentru Kubernetes. Implementarea stocării persistente pentru containere este una dintre provocările principale ale administratorilor Kubernetes, DevOps și inginerilor cloud. Containerele pot fi efemere, dar tot mai multe dintre datele lor nu sunt, așa că trebuie să se asigure supraviețuirea datelor în caz de terminare a containerului sau defecțiune hardware. Atunci când implementează containere cu Kubernetes sau aplicații containerizate, companiile realizează adesea că au nevoie de stocare persistentă. Acestea trebuie să ofere stocare rapidă și fiabilă pentru bazele de date, imaginile rădăcină și alte date utilizate de containere.
* În plus față de peisaj, Cloud Native Computing Foundation (CNCF) a publicat și alte informații despre Kubernetes Persistent Storage, inclusiv un blog care ajută la definirea modelului de stocare atașat containerului. Acest model poate fi considerat ca unul care utilizează Kubernetes în sine ca o componentă a sistemului sau serviciului de stocare.
* Container Attached Storage este un tip de stocare de date care a apărut pe măsură ce Kubernetes a câștigat proeminență. Abordarea sau modelul Container Attached Storage se bazează pe Kubernetes însuși pentru anumite capabilități, oferind în primul rând blocuri, fișiere, obiecte și interfețe la sarcinile de lucru care rulează pe Kubernetes.
* Atributele comune ale Container Attached Storage includ utilizarea extensiilor la Kubernetes, cum ar fi definițiile personalizate de resurse și utilizarea Kubernetes în sine pentru funcții care altfel ar fi dezvoltate și implementate separat pentru stocare sau gestionarea datelor. Exemplele de funcționalități furnizate de definițiile personalizate de resurse sau de Kubernetes însuși includ logica de reîncercare, furnizată de Kubernetes însuși și crearea și întreținerea unui inventar de medii și volume de stocare disponibile, livrate de obicei printr-o definiție de resurse personalizată.

**Interfață de stocare a containerelor (CSI):**

În Kubernetes versiunea 1.9, a fost introdusă versiunea inițială Alpha a Container Storage Interface (CSI). Anterior, pluginurile pentru volumul de stocare erau incluse în distribuția Kubernetes. Prin crearea unui CSI standardizat, codul necesar pentru interfața cu sistemele de stocare externe a fost separat de baza de cod de bază Kubernetes. Doar un an mai târziu, in versiunea 1.2.0, caracteristica CSI a fost făcută Generally Available (GA) în Kubernetes.

**API:**

* O componentă cheie a planului de control Kubernetes este API Server, care expune un API HTTP care poate fi invocat de alte părți ale clusterului, precum și de utilizatorii finali și componente externe. Acest API este un REST API și este de natură declarativă și este același API expus planului de control. Serverul API este susținut de etcd pentru a stoca toate înregistrările în mod persistent.
* API-ul REST în Kubernetes este un mod fundamental prin care utilizatorii și sistemele interacționează cu clusterul Kubernetes. Aceasta oferă o interfață HTTP care permite manipularea resurselor Kubernetes. Prin API-ul REST, utilizatorii pot crea, modifica, șterge și obține resurse, cum ar fi pod-uri, servicii, replicaset-uri și multe altele.
* Funcționalitățile cheie ale API-ului REST în Kubernetes includ:
* **Resurse și operații**: API-ul REST este structurat în jurul resurselor Kubernetes, cum ar fi Node, Pod, Deployment, Service etc. Fiecare resursă are un set de operații asociate, cum ar fi GET (pentru a obține informații), POST (pentru a crea o nouă resursă), PUT (pentru a actualiza o resursă existentă), PATCH (pentru a actualiza parțial o resursă) și DELETE (pentru a șterge o resursă).
* **URI-uri standard**: Fiecare resursă în Kubernetes are un URI unic asociat. De exemplu, pentru a obține toate pod-urile dintr-un anumit namespace, ai putea folosi un URI ca /api/v1/namespaces/{namespace}/pods.
* **Reprezentarea resurselor**: Resursele în Kubernetes sunt reprezentate ca obiecte JSON sau YAML. Când un utilizator face o cerere către API, răspunsul va fi de obicei în format JSON, care conține detaliile resursei solicitate.
* **Autentificare și autorizare**: API-ul REST include mecanisme de securitate pentru a controla cine poate accesa și modifica resursele Kubernetes. Autentificarea se realizează adesea prin certificate, token-uri sau alte mecanisme, iar autorizarea se bazează pe roluri și politici definite.
* **Watch API**: Kubernetes oferă o funcție "watch" pentru resurse, care permite utilizatorilor să fie notificați despre schimbările la resursele specifice. Aceasta este utilă pentru a urmări schimbările în timp real.
* **Client Libraries**: Pentru a facilita interacțiunea cu API-ul REST, Kubernetes oferă biblioteci client în mai multe limbaje de programare, cum ar fi Go, Python, Java etc., care abstractizează cererile HTTP și manipularea obiectelor.
* API-ul REST este un element central al Kubernetes, permițând dezvoltatorilor și operatorilor să automatizeze operațiunile, să integreze alte sisteme și unelte și să construiască aplicații complexe pe infrastructura Kubernetes. De exemplu, kubectl, instrumentul de linie de comandă pentru interacționarea cu clusterul Kubernetes, utilizează API-ul REST pentru a comunica cu clusterul.

**Obiecte API:**

* În Kubernetes, toate obiectele servesc drept „înregistrare a intenției” a stării clusterului și sunt capabile să definească starea dorită în care scriitorul obiectului dorește ca clusterul să fie. Ca atare, majoritatea obiectelor Kubernetes au același set. de câmpuri imbricate, după cum urmează:
* spec: Descrie starea dorită a resursei, care poate fi controlată de utilizatorii finali sau de alți controlori de nivel superior;
* stare: Descrie starea curentă a resursei, care este actualizată activ de către controlorul resursei.
* Toate obiectele din Kubernetes sunt supuse acelorași convenții API. Unele dintre acestea includ:
* Trebuie să aibă următoarele metadate sub metadatele câmpului obiect imbricat:
* **namespace:** este o modalitate de a organiza și de a separa resursele și obiectele dintr-un cluster în spații logice izolate. Este similar cu un mediu virtual care permite gruparea și izolarea resurselor în cadrul aceluiași cluster Kubernetes. Conceptul de namespace este important pentru a gestiona și a organiza resursele într-un mod care să ofere o izolare logică între diferitele aplicații sau medii de lucru din cadrul clusterului.
* Fiecare obiect în Kubernetes aparține unui namespace specific sau, dacă nu este specificat un namespace, este plasat în namespace-ul implicit numit default.
* Utilizările principale ale namespace-urilor în Kubernetes includ:

1. **Izolare logica:** Namespace-urile permit gruparea și izolarea resurselor în funcție de aplicație, mediu (cum ar fi dezvoltare, testare, producție) sau alt criteriu de organizare dorit. Aceasta ajută la evitarea interferenței între diferitele componente și aplicații care rulează în același cluster.
2. **Controlul accesului:** Namespace-urile pot fi folosite pentru a gestiona accesul și permisiunile utilizatorilor sau serviciilor la resursele din cluster.
3. **Limitarea resurselor:** Este posibil să se aplice limitări de resurse (CPU, memorie etc.) la nivel de namespace, pentru a asigura că resursele sunt distribuite și utilizate în mod corespunzător între diferitele medii sau aplicații.
4. **Etichetare și organizare:** Namespace-urile pot fi etichetate și folosite pentru a organiza și a filtra resursele în funcție de anumite criterii, facilitând gestionarea și monitorizarea acestora.
5. **Managementul ciclului de viață:** Namespace-urile facilitează gestionarea și eliminarea resurselor asociate unei anumite aplicații sau medii de lucru, permițând unui namespace să fie eliminat împreună cu toate resursele sale.

* **nume**: un șir care identifică în mod unic obiectul în spațiul de nume definit;
* **uid**: un șir unic care este capabil să facă distincția între obiectele cu același nume în spațiu și timp (chiar și între ștergeri și recreări cu același nume).
* Poate fi gestionat de un alt controlor, care este definit în câmpul metadata.ownerReferences:
* Cel mult un alt obiect va fi controlerul de gestionare al obiectului controlat, care este definit de câmpul controller.
* Poate fi colectat gunoiul dacă proprietarul este șters:
* Când un obiect este șters, toate obiectele dependente pot fi, de asemenea, șterse în cascadă.

**Resurse personalizate, controlori și operatori:**

* API-ul Kubernetes poate fi extins folosind Resurse personalizate, care reprezintă obiecte care nu fac parte din instalarea standard Kubernetes. Aceste resurse personalizate sunt declarate folosind definiții personalizate de resurse (CRD), care este un fel de resursă care poate fi înregistrată și dezînregistrată dinamic fără a închide sau a reporni un cluster care rulează în prezent.
* Controlerele personalizate sunt un alt mecanism de extensie care interacționează cu API-ul Kubernetes, similar controlerelor implicite din managerul de controlere Kubernetes preinstalat standard. Acești controlori pot interacționa cu resurse personalizate pentru a permite un API declarativ: utilizatorii pot declara starea dorită a lumii prin intermediul resurselor personalizate și este responsabilitatea controlorului personalizat să observe modificarea și să o reconcilieze.
* Combinația de resurse personalizate și controlere personalizate este adesea denumită operator Kubernetes. Cazul de utilizare cheie pentru operatori este de a capta scopul unui operator uman care gestionează un serviciu sau un set de servicii și de a le implementa folosind automatizarea și cu un API declarativ care sprijină această automatizare. Operatorii umani care se ocupă de aplicații și servicii specifice au cunoștințe profunde despre cum ar trebui să se comporte sistemul, cum să-l implementeze și cum să reacționeze în cazul în care există probleme.
* Exemple de probleme rezolvate de operatori includ realizarea și restabilirea copiilor de siguranță ale stării aplicației respective și gestionarea actualizărilor codului aplicației alături de modificări aferente, cum ar fi schemele bazei de date sau setările de configurare suplimentare. Mai multe proiecte notabile din cadrul programului de incubare al Cloud Native Computing Foundation urmează modelul operatorului pentru a extinde Kubernetes, inclusiv Argo, Open Policy Agent și Istio.

**Securitate API:**

* Kubernetes definește următoarele strategii pentru controlul accesului la API-ul său:

**Securitatea transporturilor (blocuri de retea):**

* Serverul API Kubernetes ascultă pe un port TCP care servește trafic HTTPS, pentru a impune securitatea stratului de transport (TLS) folosind certificate semnate de catre Certificate Autority-ului (CA) intern al cluster-ului.
* În versiunile mai vechi de Kubernetes, serverul API a acceptat ascultarea atât pe porturile HTTP, cât și pe porturile HTTPS (numărul portului HTTP nu avea niciun fel de securitate de transport). Acest lucru a fost depreciat în v1.10 și în cele din urmă a renunțat la suportul în v1.20 a Kubernetes.

**Autentificare:**

* Toate solicitările făcute către serverul API Kubernetes sunt de așteptat să fie autentificate și acceptă mai multe strategii de autentificare, dintre care unele sunt enumerate mai jos:

1. certificate client X.509
2. Token de purtător
3. Token al cont de serviciu, destinate accesului API programatic

* De obicei, se așteaptă ca utilizatorii să indice și să definească detaliile URL ale clusterului împreună cu acreditările necesare într-un fișier kubeconfig, care sunt acceptate nativ de alte instrumente Kubernetes, cum ar fi kubectl și bibliotecile oficiale de clienți Kubernetes.

**Autorizare:**

API-ul Kubernetes acceptă următoarele moduri de autorizare:

1. Modul de autorizare a nodului: acordă o listă fixă ​​de operațiuni de solicitări API pe care demonii kubele au voie să le efectueze, pentru a funcționa corect.
2. Modul de control al accesului bazat pe atribute (ABAC): acordă drepturi de acces utilizatorilor prin utilizarea politicilor de control al accesului definite care combină atributele împreună.
3. Modul de control al accesului bazat pe roluri (RBAC): acordă drepturi de acces utilizatorilor pe baza rolurilor care sunt acordate utilizatorului, unde fiecare rol definește o listă de acțiuni care sunt permise.
4. Modul Webhook: Interogează un serviciu API REST pentru a determina dacă un utilizator este autorizat să efectueze o anumită acțiune (Admission Controller).

**Clienți API:**

Kubernetes acceptă mai mulți clienți API distribuiti de producatorul platformei (officeal clients) sau clienti dezvoltati de comunicatate Open Source:

1. kubelet: linie de comandă pentru interacțiunea cu planul de control Kubernetes
2. Biblioteci client oficiale întreținute de Kubernetes pentru C, .NET, Go, Haskell, Java, JavaScript, Perl, Python și Ruby

**Cluster API:**

Aceleași principii de proiectare API au fost utilizate pentru a defini un API pentru a crea, configura și gestiona în mod programatic clusterele Kubernetes. Aceasta se numește Cluster API. Un concept cheie încorporat în API este utilizarea infrastructurii ca software sau ideea că infrastructura clusterului Kubernetes este ea însăși o resursă/obiect care poate fi gestionat la fel ca orice alte resurse Kubernetes. În mod similar, mașinile care alcătuiesc clusterul sunt, de asemenea, tratate ca o resursă Kubernetes. API-ul are două componente – API-ul de bază și o implementare a dezvoltatorului. Implementarea furnizorului constă în funcții specifice furnizorului de cloud care îi permit lui Kubernetes să furnizeze API-ul cluster într-un mod care este bine integrat cu serviciile și resursele furnizorului de cloud.

**Adăugarea de resurse personalizate:**

* Kubernetes oferă două moduri de a adăuga resurse personalizate la clusterul dvs.:

1. CRD-urile sunt simple și pot fi create fără nicio programare.
2. Agregarea API necesită programare, dar permite mai mult control asupra comportamentelor API, cum ar fi modul în care sunt stocate datele și conversia între versiunile API.

* Kubernetes oferă aceste două opțiuni pentru a satisface nevoile diferiților utilizatori, astfel încât nici ușurința de utilizare, nici flexibilitatea să nu fie compromise.
* API-urile agregate sunt servere API subordonate care stau în spatele serverului API principal, care acționează ca un proxy. Acest aranjament se numește API Aggregation (AA). Pentru utilizatori, API-ul Kubernetes pare extins.
* CRD-urile permit utilizatorilor să creeze noi tipuri de resurse fără a adăuga un alt server API. Nu trebuie să înțelegeți agregarea API pentru a utiliza CRD-uri.
* Indiferent de modul în care sunt instalate, noile resurse sunt denumite resurse personalizate pentru a le diferenția de resursele Kubernetes încorporate (cum ar fi pod-urile).

**Alegerea unei metode de adăugare a resurselor personalizate:**

* CRD-urile sunt mai ușor de utilizat. API-urile agregate sunt mai flexibile. Alege metoda care corespunde cel mai bine nevoilor tale.
* De obicei, CRD-urile sunt potrivite dacă:

1. Ai o mână de câmpuri
2. Utilizați resursa în cadrul companiei dvs. sau ca parte a unui mic proiect open-source (spre deosebire de un produs comercial)

**Compararea ușurinței de utilizare:**

API-ul agregat vs CRD:

1. Nu necesită programare. Utilizatorii pot alege orice limbă pentru un controler CRD. Necesită programare și construirea binare și imagini.
2. Niciun serviciu suplimentar de rulat; CRD-urile sunt gestionate de serverul API. Un serviciu suplimentar de creat și care ar putea eșua.
3. Nu există suport continuu odată ce CRD este creat. Orice remediere a erorilor este preluată ca parte a upgrade-urilor Kubernetes Master normale. Poate fi necesar să colectați periodic remedieri de erori din amonte și să reconstruiți și să actualizați serverul API agregat.
4. Nu este nevoie să gestionați mai multe versiuni ale API-ului dvs.; de exemplu, când controlați clientul pentru această resursă, îl puteți actualiza în sincronizare cu API-ul. Trebuie să gestionați mai multe versiuni ale API-ului dvs.; de exemplu, atunci când dezvoltați o extensie pentru a partaja cu lumea.

**Trasaturi comune:**

Caracteristică si ce face:

CRUD: Noile puncte finale acceptă operațiuni de bază CRUD prin HTTP și kubectl

Watch: Noile puncte finale acceptă operațiunile Kubernetes Watch prin HTTP

Discovery: Clienții precum kubectl și tabloul de bord oferă automat operațiuni de listă, afișare și editare în câmp pentru resursele dvs

json-patch: Noile puncte finale acceptă PATCH cu Content-Type: application/json-patch+json

merge-patch: Noile puncte finale acceptă PATCH cu Content-Type: application/merge-patch+json

HTTPS: Noile puncte finale utilizează HTTPS

Autentificare încorporată: Accesul la extensie folosește serverul API de bază (stratul de agregare) pentru autentificare

Autorizare încorporată: Accesul la extensie poate reutiliza autorizarea utilizată de serverul API de bază; de exemplu, RBAC.

Finalizatoare: Blocați ștergerea resurselor extensiei până când are loc curățarea externă.

Admitere Webhooks: permițând extinderea și personalizarea comportamentului clusterului în ambele cazuri.

UI/CLI Display: Kubectl, tabloul de bord poate afișa resurse de extensie.

Unset versus Empty: Clienții pot distinge câmpurile nesetate de câmpurile cu valoare zero.

Generarea bibliotecilor client: Kubernetes oferă biblioteci client generice, precum și instrumente pentru a genera biblioteci client specifice tipului.

Etichete și adnotări: Metadate comune pentru obiecte pe care instrumentele știu să le editeze pentru resurse de bază și personalizate.

**Un exemplu de manifest pentru un computer de bază care utilizează Component Runtime Descriptor (CRD) poate arăta astfel:**

apiVersion: hardware.example.com/v1

kind: Computer

metadata:

name: computer-de-baza

spec:

cpu:

model: Intel i5

cores: 4

memory:

size: 8GB

type: DDR4

storage:

- type: SSD

size: 256GB

network:

- type: Ethernet

speed: 1Gbps

operatingSystem:

name: Ubuntu

version: "20.04"

**Acest manifest descrie un computer de bază, cu următoarele specificații:**

* Un procesor Intel i5 cu 4 nuclee.
* 8GB memorie RAM de tip DDR4.
* Un disc SSD de 256GB.
* O conexiune de rețea Ethernet cu viteză de 1Gbps.
* Sistemul de operare instalat este Ubuntu 20.04.
* Acest manifest este doar un exemplu și poate fi adaptat în funcție de nevoile specifice ale sistemului pe care doriți să-l descrieți.
* Componentele pot varia în funcție de configurația dorită, iar câmpurile pot fi modificate sau extinse pentru a se potrivi cu cerințele specifice.

**Modelul operatorului:**

* Operatorii sunt extensii software pentru Kubernetes care folosesc resurse personalizate pentru a gestiona aplicațiile și componentele acestora. Operatorii urmează principiile Kubernetes, în special bucla de control.
* Conceptul de operatori în Kubernetes a fost introdus de către compania CoreOS în 2016. Aceștia au dezvoltat conceptul și au lansat apoi framework-ul Operator SDK în 2018, care a facilitat dezvoltatorilor să creeze și să gestioneze operatori pentru aplicații Kubernetes mai ușor și mai eficient. De atunci, conceptul de operatori a continuat să crească în popularitate și a devenit un element important în gestionarea aplicațiilor complexe pe platforma Kubernetes.

**Motivația:**

* Modelul operatorului urmărește să surprindă scopul cheie al unui operator uman care gestionează un serviciu sau un set de servicii. Operatorii umani care se ocupă de aplicații și servicii specifice au cunoștințe profunde despre cum ar trebui să se comporte sistemul, cum să-l implementeze și cum să reacționeze în cazul în care există probleme.
* Oamenilor care rulează sarcini de lucru pe Kubernetes le place adesea să folosească automatizarea pentru a se ocupa de sarcini repetabile. Modelul operator surprinde modul în care puteți scrie cod pentru a automatiza o sarcină dincolo de ceea ce oferă Kubernetes însuși.

**Operatori în Kubernetes:**

* Kubernetes este conceput pentru automatizare. Din cutie, obțineți o mulțime de automatizări încorporate din nucleul Kubernetes. Puteți folosi Kubernetes pentru a automatiza implementarea și rularea sarcinilor de lucru și puteți automatiza modul în care Kubernetes face asta.
* Conceptul de model de operator al Kubernetes vă permite să extindeți comportamentul clusterului fără a modifica codul Kubernetes însuși prin conectarea controlerelor la una sau mai multe resurse personalizate. Operatorii sunt clienți ai API-ului Kubernetes care acționează ca controlori pentru o resursă personalizată.

**Un exemplu de operator:**

Unele dintre lucrurile pe care le puteți utiliza un operator pentru a le automatiza includ:

1. implementarea unei aplicații la cerere
2. luarea și restaurarea copiilor de rezervă ale stării respectivei aplicații
3. gestionarea actualizărilor codului aplicației împreună cu modificările aferente, cum ar fi schemele bazei de date sau setările de configurare suplimentare
4. publicarea unui serviciu în aplicațiile care nu acceptă API-urile Kubernetes pentru a le descoperi
5. simularea eșecului în întregul cluster sau în parte a acestuia pentru a-i testa rezistența
6. alegerea unui lider pentru o cerere distribuită fără un proces intern de alegere a membrilor

Cum ar putea arăta un operator mai detaliat? Iată un exemplu:

1. resursă personalizată numită SampleDB, pe care o puteți configura în cluster.
2. implementare care se asigură că rulează un Pod care conține partea de control a operatorului.
3. imagine de container a codului operatorului.
4. Cod de controler care interogează planul de control pentru a afla ce resurse SampleDB sunt configurate.
5. Miezul operatorului este codul pentru a spune serverului API cum să facă realitatea să se potrivească cu resursele configurate.
6. Dacă adăugați un nou SampleDB, operatorul setează PersistentVolumeClaims pentru a oferi stocare durabilă a bazei de date, un StatefulSet pentru a rula SampleDB și un Job pentru a gestiona configurația inițială.
7. Dacă îl ștergeți, operatorul face un instantaneu, apoi se asigură că StatefulSet și Volumes sunt, de asemenea, eliminate.
8. Operatorul gestionează, de asemenea, copiile de rezervă regulate ale bazei de date. Pentru fiecare resursă SampleDB, operatorul stabilește când să creeze un Pod care se poate conecta la baza de date și poate face copii de rezervă. Aceste Pod-uri s-ar baza pe un ConfigMap și/sau un Secret care are detalii de conectare la baza de date și acreditări.
9. Deoarece operatorul își propune să ofere o automatizare robustă pentru resursa pe care o gestionează, ar exista cod suplimentar de suport. Pentru acest exemplu, codul verifică dacă baza de date rulează o versiune veche și, dacă da, creează obiecte Job care o actualizează pentru tine.

**Desfășurarea operatorilor:**

Cea mai obișnuită modalitate de a implementa un operator este să adăugați definiția personalizată a resurselor și controlerul asociat acestuia la cluster. Controlerul va rula în mod normal în afara planului de control, la fel cum ați rula orice aplicație containerizată. De exemplu, puteți rula controlerul în clusterul dvs. ca implementare.

**Folosirea un operator:**

* Odată ce ați instalat un operator, îl veți folosi prin adăugarea, modificarea sau ștergerea tipului de resursă pe care o folosește operatorul. Urmând exemplul de mai sus, ați configura o Implementare pentru operatorul însuși și apoi:
* kubectl obține SampleDB # găsi baze de date configurate
* kubectl edit SampleDB/example-database # modifica manual unele setări
* Operatorul se va ocupa de aplicarea modificărilor, precum și de menținerea serviciului existent în stare bună.

**Scrierea propriului operator:**

* Dacă nu există un operator în ecosistem care să implementeze comportamentul dorit, îl puteți codifica pe al dvs.
* De asemenea, implementați un operator (adică un controler) folosind orice limbă / timp de rulare care poate acționa ca client pentru API-ul Kubernetes.
* Următoarele sunt câteva biblioteci și instrumente pe care le puteți folosi pentru a vă scrie propriul operator nativ în cloud:

1. Operator SDK (Scris in golang)
2. Cadrul Operatorului Charmed (solutie proprietara furnizata de Canonical)
3. Java Operator SDK
4. Kopf (Cadru Pythonic al operatorului Kubernetes)
5. kube-rs
6. kubebuilder
7. KubeOps (SDK operator .NET)
8. KUDO (operator declarativ universal Kubernetes)
9. Catarg
10. Metacontroller împreună cu WebHooks pe care le implementați singur
11. Cadrul operatorului
12. operator-shell

***Implementarea, folosind KOPF, unui operator care sa faca deployment-ul unei imagini wordpress ce foloseste MariaDB Gallera pentru stocare***

**Descrierea principala a implementarii:**

**Pentru a implementa un operator Kubernetes care să facă deployment-ul unei imagini WordPress ce folosește MariaDB Galera pentru stocare, va trebui să urmezi mai mulți pași și să folosești diferite resurse ale Kubernetes. Iată o descriere generală a procesului pe care îl poți urma, având în vedere imaginea încărcată:**

Crearea imaginii Docker pentru WordPress și MariaDB Galera**:**

**Începe prin a crea o imagine Docker personalizată pentru WordPress care să poată comunica cu o bază de date MariaDB Galera.**

**De asemenea, ai nevoie de o imagine Docker pentru clusterul MariaDB Galera, dacă nu există una disponibilă deja.**

Configurarea Persistent Volumes (PV) și Persistent Volume Claims (PVC)**:**

**Definește PV-uri pentru a stoca datele persistente pentru WordPress și baza de date.**

**Creează PVC-uri care vor fi asociate cu aceste PV-uri. În Kubernetes, PVC-urile sunt folosite de pod-uri pentru a solicita spațiu de stocare fizic.**

Crearea unui ConfigMap**:**

**ConfigMap este folosit pentru a păstra și gestiona configurația aplicației, cum ar fi fișierele de configurare și variabilele de mediu.**

Crearea unui Secret**:**

**Secret este utilizat pentru a stoca și a gestiona informațiile sensibile, cum ar fi parolele și cheile API.**

Definirea și crearea obiectelor Kubernetes pentru WordPress și baza de date**:**

Deployment**: Creează un deployment pentru WordPress care să utilizeze imaginea Docker creată. Configurația deployment-ului va include referințe la PVC-uri, ConfigMap și Secret.**

Service**: Creează un serviciu Kubernetes pentru a expune WordPress la traficul de rețea.**

StatefulSet sau Deployment pentru MariaDB Galera**: Creează un StatefulSet (recomandat pentru baze de date) sau un Deployment pentru clusterul MariaDB Galera, asigurându-te că are stocare persistentă și este configurat corespunzător pentru replicare.**

Implementarea Operatorului**:**

**Scrie un operator Kubernetes care să gestioneze toate aceste resurse și să automatizeze deployment-ul WordPress cu MariaDB Galera.**

**Operatorul va trebui să monitorizeze starea clusterului, să facă upgrade-uri, să gestioneze backup-uri și să asigure scalarea automată dacă este necesar.**

**Pentru a scrie un operator, poți folosi SDK-ul Operator Framework sau poți scrie propriul operator folosind clientul Go pentru Kubernetes. Operatorul va trebui să implementeze logica specifică pentru a gestiona ciclul de viață al aplicației WordPress și al bazei de date MariaDB Galera în cadrul clusterului Kubernetes.**

**Implementarea docker image, un cointainer docker pentru wordpress si PHP\_FPM, cu un port 80 expus:**

Pentru a crea o imagine Docker pentru WordPress care rulează PHP-FPM cu portul 80 expus, puteți urma pașii de mai jos. Implementarea se bazează pe crearea unui Dockerfile care definește cum ar trebui să fie construit containerul.

1. Creem un fișier Dockerfile: Deschideți un editor de text și creați un nou fișier numit Dockerfile cu următorul conținut:

dockerfile

# Folosim imaginea oficială WordPress cu PHP-FPM

FROM wordpress:php7.4-fpm

# Instalăm nginx pentru a servi paginile

RUN apt-get update && apt-get install -y nginx && rm -rf /var/lib/apt/lists/\*

# Configurăm nginx

COPY default.conf /etc/nginx/conf.d/default.conf

# Copiem un script care va rula atât nginx cât și PHP-FPM

COPY entrypoint.sh /entrypoint.sh

RUN chmod +x /entrypoint.sh

# Expunem portul 80

EXPOSE 80

# Executăm scriptul de entrypoint

ENTRYPOINT ["/entrypoint.sh"]

1. Creem un fișier de configurare nginx: Creem un fișier numit default.conf pentru configurarea nginx. Acesta va servi ca server web în fața procesului PHP-FPM. Exemplu de conținut pentru default.conf:

nginx

server {

listen 80;

server\_name localhost;

root /var/www/html;

index index.php;

location / {

try\_files $uri $uri/ /index.php?$args;

}

location ~ \.php$ {

fastcgi\_pass wordpress:9000;

fastcgi\_index index.php;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

include fastcgi\_params;

}

}

1. Creem un script de entrypoint: Acest script va porni atât nginx cât și PHP-FPM. Salvați-l ca entrypoint.sh:

bash

#!/bin/bash

# Pornim PHP-FPM

php-fpm

# Pornim nginx în foreground

nginx -g 'daemon off;'

1. Construiți imaginea Docker: Deschideți un terminal în directorul în care se află Dockerfile, default.conf, și entrypoint.sh, și rulați comanda de build:

bash

docker build -t wordpress-custom .

1. Rulați containerul Docker: După construirea imaginii, rulați containerul folosind:

bash

docker run -d -p 80:80 --name wordpress-server wordpress-custom

Acum ar trebui să aveți un container Docker care rulează WordPress cu PHP-FPM și nginx, cu portul 80 expus. Puteți accesa aplicația navigând către http://localhost într-un browser web.

**Fișier YAML pentru Deployment cu resurse limitate și rezervate:**

Pentru a crea un fișier YAML de Deployment în Kubernetes pentru o aplicație WordPress, trebuie să specificați configurația containerelor, inclusiv resursele (requests și limits) pentru CPU și memorie. Mai jos este un exemplu de manifest Kubernetes care definește un Deployment pentru WordPress cu resursele specificate:

apiVersion: apps/v1

kind: Deployment

metadata:

name: wordpress

labels:

app: wordpress

spec:

replicas: 1

selector:

matchLabels:

app: wordpress

template:

metadata:

labels:

app: wordpress

spec:

containers:

- name: wordpress

image: wordpress:latest

ports:

- containerPort: 80

env:

- name: WORDPRESS\_DB\_HOST

value: wordpress-db

- name: WORDPRESS\_DB\_USER

value: wordpressuser

- name: WORDPRESS\_DB\_PASSWORD

valueFrom:

secretKeyRef:

name: mysql-pass

key: password

- name: WORDPRESS\_DB\_NAME

value: wordpress

resources:

requests:

memory: "512Mi"

cpu: "250m"

limits:

memory: "1024Mi"

cpu: "500m"

volumes:

- name: wordpress-persistent-storage

persistentVolumeClaim:

claimName: wordpress-pvc

---

apiVersion: v1

kind: Service

metadata:

name: wordpress

labels:

app: wordpress

spec:

ports:

- port: 80

selector:

app: wordpress

type: LoadBalancer

În acest exemplu, se face deploy la un container cu imaginea wordpress:latest. Portul 80 al containerului este expus și este legat de un serviciu de tip LoadBalancer, care va expune aplicația WordPress la internet (într-un mediu de cloud).

La secțiunea resources, am definit cereri minime (requests) de 512Mi de memorie și 250m CPU, ceea ce înseamnă că Kubernetes va asigura că containerul va avea aceste resurse disponibile înainte de a fi rulat. Am definit, de asemenea, limite (limits) de 1024Mi de memorie și 500m CPU, ceea ce înseamnă că containerul nu va utiliza mai multe resurse decât aceste limite.

Pentru detalii de conectare la baza de date, se utilizează variabile de mediu, unde WORDPRESS\_DB\_PASSWORD este setat printr-o referință la un obiect Kubernetes Secret care ar trebui creat separat pentru a stoca parole în mod sigur.

Acest manifest poate fi creat și aplicat în cluster-ul Kubernetes cu ajutorul următoarelor comenzi:

kubectl create -f wordpress-deployment.yaml

Crearea unui Persistent Volume Claim (PVC) și a unui Persistent Volume (PV) pentru a stoca datele WordPress și MariaDB:

**Pentru a crea un Persistent Volume Claim (PVC) în Kubernetes, vom avea nevoie de un fișier YAML care descrie resursele și specificațiile necesare. Acesta va permite ca datele să fie persistente, chiar și atunci când containerele care rulează aplicațiile sunt înlocuite sau reîmprospătate.**

**apiVersion: v1**

**kind: PersistentVolumeClaim**

**metadata:**

**name: wordpress-pvc**

**spec:**

**accessModes:**

**- ReadWriteOnce**

**resources:**

**requests:**

**storage: 10Gi**

**storageClassName: standard**

**În acest exemplu:**

* **apiVersion: v1 specifică versiunea API pe care o folosim pentru acest obiect.**
* **kind: PersistentVolumeClaim este tipul de obiect pe care îl creăm.**
* **metadata conține metadate despre PVC, cum ar fi name: wordpress-pvc, care este numele PVC-ului.**
* **spec conține specificațiile pentru PVC:**
  + **accessModes: - ReadWriteOnce indică faptul că volumul va putea fi montat ca read-write de un singur nod. Alte opțiuni includ ReadOnlyMany și ReadWriteMany.**
  + **resources: requests: storage: 10Gi specifică cererea de spațiu de stocare pentru PVC, în acest caz, 10 GB.**
  + **storageClassName: standard se referă la clasa de stocare care să fie folosită pentru PVC. Clasele de stocare sunt configurate de administratorul cluster-ului și pot avea diferite politici, cum ar fi performanța, replicarea și backup-ul. Numele standard este adesea utilizat pentru clasa de stocare implicită în multe cluster-e Kubernetes, dar acest nume poate varia în funcție de configurația cluster-ului dvs.**

**După ce ați creat fișierul YAML, puteți aplica acest manifest în cluster-ul Kubernetes folosind comanda kubectl:**

**bash**

**kubectl apply -f pvc-wordpress.yaml**

**Înlocuiți pvc-wordpress.yaml cu numele fișierului în care ați salvat manifestul PVC.**

**PVC-ul creat va fi gestionat de Kubernetes, iar când un pod va solicita acest PVC prin numele său, Kubernetes va încerca să îl monteze pe pod în funcție de disponibilitatea și conformitatea PV-urilor din cluster. Acest lucru permite WordPress-ului să păstreze datele pe disc peste reîncărcările și rearanjările pod-urilor.**

**Implementarea Ingress și TLS:**

Pentru a configura un Ingress în Kubernetes care să direcționeze traficul către aplicația dvs. WordPress și să folosească TLS pentru securizarea conexiunii, aveți nevoie de două componente principale:

Un certificat TLS - puteți obține unul de la o Autoritate de Certificare (CA) sau puteți genera un certificat auto-semnat pentru testare.

Un fișier YAML pentru Ingress - care definește regulile de rutare a traficului și referințează certificatul TLS.

Dacă folosiți cert-manager pentru gestionarea certificatelor în Kubernetes, acesta poate automatiza crearea și reînnoirea certificatelor pentru Ingress. Mai jos este un exemplu care presupune că aveți cert-manager instalat și configurat:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: wordpress-ingress

annotations:

cert-manager.io/cluster-issuer: "letsencrypt-prod"

nginx.ingress.kubernetes.io/force-ssl-redirect: "true"

spec:

tls:

- hosts:

- your-domain.com

secretName: wordpress-tls

rules:

- host: your-domain.com

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: wordpress

port:

number: 80

În acest fișier YAML:

apiVersion: networking.k8s.io/v1 - se referă la versiunea API-ului de networking folosit.

kind: Ingress - specifică că obiectul este de tip Ingress.

metadata - conține metadate despre Ingress.

annotations - conține diverse setări:

cert-manager.io/cluster-issuer - specifică emițătorul pentru certificatul TLS, în acest caz letsencrypt-prod, care ar trebui să fie un ClusterIssuer predefinit în cert-manager.

nginx.ingress.kubernetes.io/force-ssl-redirect - forțează redirecționarea traficului de la HTTP la HTTPS.

spec - descrie regulile și configurarea TLS pentru Ingress.

tls - specifică configurația TLS:

hosts - lista domeniilor pentru care este valid certificatul.

secretName - numele Secret-ului Kubernetes unde cert-manager va stoca certificatul TLS.

rules - definește regulile de rutare a traficului către serviciile din backend.

Pentru a aplica acest fișier, salvați-l ca wordpress-ingress.yaml și rulați:

kubectl apply -f wordpress-ingress.yaml

Pentru a crea un Secret manual pentru un certificat TLS, utilizați comanda:

kubectl create secret tls wordpress-tls --cert=path/to/tls.crt --key=path/to/tls.key

**Implementarea Serviciului:**

Pentru a expune aplicația WordPress în interiorul unui cluster Kubernetes, veți avea nevoie să creați un obiect Service. Service-ul va defini cum alte aplicații din cluster sau utilizatori externi pot accesa aplicația dvs. WordPress. Aici este un exemplu de fișier YAML pentru un Service care expune WordPress:

yaml

apiVersion: v1

kind: Service

metadata:

name: wordpress-service

labels:

app: wordpress

spec:

type: ClusterIP

ports:

- port: 80

targetPort: 80

protocol: TCP

selector:

app: wordpress

În acest fișier:

* apiVersion: v1 indică versiunea API-ului utilizat pentru a crea Service.
* kind: Service definește tipul de obiect Kubernetes ca fiind un Service.
* metadata conține metadatele pentru Service, inclusiv name și labels.
* spec specifică configurația Service:
  + type: ClusterIP indică faptul că acesta este un Service care va fi accesibil intern în cluster. Alte opțiuni pot include NodePort sau LoadBalancer pentru acces extern.
  + ports definește lista de porturi pentru care Service-ul va face proxy. port este portul pe care Service-ul îl va expune, iar targetPort este portul containerului la care Service-ul va redirecționa traficul.
  + selector specifică care Pods vor fi expuse prin acest Service. Se potrivește cu labels setate pe Pods.

Salvam fișierul YAML și aplicați-l în cluster-ul Kubernetes utilizând următoarea comandă:

kubectl apply -f wordpress-service.yaml

**MariaDB Galera - Este specificată utilizarea bazei de date MariaDB cu Galera pentru replicarea cluster**:

Pentru a rula un cluster MariaDB Galera în Kubernetes, va trebui să configurăm mai multe componente. În esență, acestea includ:

1. Un StatefulSet sau Deployment pentru MariaDB Galera noduri.
2. Un PersistentVolume (PV) sau PersistentVolumeClaim (PVC) pentru persistența datelor.
3. Un Service pentru a expune și echilibra conexiunile către nodurile MariaDB.

Înainte de a începe, trebuie să vă asigurați că aveți configurată stocarea persistentă în cluster-ul dvs. Kubernetes, astfel încât datele să supraviețuiască repornirilor și reschedulărilor containerelor.

Iată un exemplu simplificat de YAML care configurează un cluster MariaDB Galera folosind StatefulSet în Kubernetes. Acest exemplu nu include toate detaliile posibile, ci oferă o configurație de bază pentru a vă ajuta să începem.

apiVersion: v1

kind: Service

metadata:

name: mariadb-galera

spec:

ports:

- port: 3306

clusterIP: None

selector:

app: mariadb-galera

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: mariadb-galera

spec:

serviceName: "mariadb-galera"

replicas: 3

selector:

matchLabels:

app: mariadb-galera

template:

metadata:

labels:

app: mariadb-galera

spec:

containers:

- name: mariadb

image: mariadb:10.4

args:

- "--wsrep-new-cluster"

- "--wsrep-on=ON"

- "--wsrep-sst-method=rsync"

env:

- name: MYSQL\_ROOT\_PASSWORD

valueFrom:

secretKeyRef:

name: mariadb-secret

key: password

- name: MYSQL\_DATABASE

value: wordpressdb

- name: MYSQL\_USER

value: wordpressuser

- name: MYSQL\_PASSWORD

valueFrom:

secretKeyRef:

name: mariadb-secret

key: user-password

ports:

- containerPort: 3306

volumeMounts:

- name: data

mountPath: /var/lib/mysql

volumeClaimTemplates:

- metadata:

name: data

spec:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 10Gi

În acest exemplu:

* Un **Service headless** este creat pentru a controla rețeaua între nodurile Galera.
* Un **StatefulSet** este folosit pentru a menține identitatea fiecărui nod Galera. StatefulSet este ideal pentru baze de date și alte aplicații care necesită o stocare persistentă unică pentru fiecare instanță.
* Variabilele de mediu precum MYSQL\_ROOT\_PASSWORD, MYSQL\_DATABASE, MYSQL\_USER și MYSQL\_PASSWORD sunt setate folosind referințe la un Kubernetes Secret care trebuie creat separat pentru a stoca aceste valori în mod sigur.
* Directorul de date al MariaDB este montat la o **PersistentVolume** prin **volumeClaimTemplates** care asigură fiecărui nod din clusterul Galera spațiul său de stocare persistent.

Pentru a crea secretul necesar pentru variabilele de mediu mariadb-secret, folosim comanda:

kubectl create secret generic mariadb-secret \

--from-literal=password='your-root-password' \

--from-literal=user-password='your-user-password'

După ce am configurat YAML-ul și ați creat Secret-ul, puteți aplica fișierul în cluster-ul dvs. Kubernetes folosind:

kubectl apply -f mariadb-galera.yaml

**CRD-uri (namespace) din Mariadb database:**

Pentru a implementa un operator personalizat în Kubernetes folosind KOPF (Kubernetes Operator Pythonic Framework), care să gestioneze deployment-ul unei aplicații WordPress ce utilizează MariaDB Galera pentru stocare, trebuie să definim mai întâi un Custom Resource Definition (CRD) pentru a descrie resursa specifică pe care operatorul nostru o va gestiona. În acest caz, putem crea un CRD pentru o instanță a bazei de date MariaDB Galera.

Acest CRD va servi drept "șablon" pentru crearea instanțelor de baze de date prin intermediul operatorului nostru:

apiVersion: apiextensions.k8s.io/v1

kind: CustomResourceDefinition

metadata:

name: mariadbgaleraclusters.mycompany.com

spec:

group: mycompany.com

names:

kind: MariaDBGaleraCluster

listKind: MariaDBGaleraClusterList

plural: mariadbgaleraclusters

singular: mariadbgaleracluster

scope: Namespaced

versions:

- name: v1

served: true

storage: true

schema:

openAPIV3Schema:

type: object

properties:

spec:

type: object

properties:

replicas:

type: integer

minimum: 3

image:

type: string

database:

type: object

properties:

name:

type: string

user:

type: string

passwordSecret:

type: object

properties:

name:

type: string

key:

type: string

storage:

type: object

properties:

size:

type: string

storageClassName:

type: string

required: [replicas, image, database, storage]

required: [spec]

Acest CRD definește o nouă resursă numită MariaDBGaleraCluster în grupul mycompany.com cu versiunea v1. Schema specifică câteva proprietăți esențiale pentru a configura o instanță MariaDB Galera, cum ar fi numărul de replici, imaginea containerului, detaliile bazei de date (inclusiv un secret pentru parolă) și configurația de stocare.

După ce definim CRD-ul, trebuie să îl aplicați în cluster-ul Kubernetes:

kubectl apply -f mariadbgaleracluster-crd.yaml

Odată ce CRD-ul este creat, puteți începe dezvoltarea operatorului KOPF care va asculta evenimente pentru crearea, actualizarea sau ștergerea resurselor MariaDBGaleraCluster și va efectua acțiunile corespunzătoare, cum ar fi deploy-ul clusterului MariaDB Galera în Kubernetes.

În ceea ce privește implementarea propriu-zisă a operatorului cu KOPF, va implica scrierea codului Python care să interacționeze cu API-ul Kubernetes pentru a gestiona ciclul de viață al instanțelor MariaDB Galera conform specificațiilor din resursele MariaDBGaleraCluster. Aceasta este o sarcină mai avansată și presupune o înțelegere solidă a interfețelor programatice ale Kubernetes, precum și a framework-ului KOPF.

**Fisierul yaml pentru User Secret din MariaDB database:**

Pentru a gestiona accesul la baza de date MariaDB Galera în cadrul implementării unui operator cu KOPF, este esențial să stocăm credențialele de acces într-un mod sigur. Kubernetes oferă obiectul Secret pentru acest scop, care poate fi utilizat pentru a stoca și gestiona informații sensibile, cum ar fi parole, OAuth tokens și ssh keys.

Iată un exemplu de fișier YAML pentru a crea un Kubernetes Secret, care conține utilizatorul și parola pentru accesul la o instanță a bazei de date MariaDB. Acest Secret poate fi referit apoi în definitiile resurselor (de exemplu, în StatefulSet pentru MariaDB Galera sau în Deployment pentru WordPress), asigurând astfel că datele sensibile nu sunt expuse direct în fișierele de configurare sau în sursa codului.

apiVersion: v1

kind: Secret

metadata:

name: mariadb-user-secret

namespace: default

type: Opaque

data:

username: d29yZHByZXNzdXNlcg== # Base64 encoded value for "wordpressuser"

password: cGFzc3dvcmQxMjM= # Base64 encoded value for "password123"

Pentru a crea acest Secret, înlocuiți valorile username și password cu valorile dorite, codificate în Base64. Puteți folosi comanda echo -n 'yourtext' | base64 pentru a obține valoarea codată a textului dvs.

De exemplu, pentru a codifica textul "wordpressuser", utilizați:

echo -n 'wordpressuser' | base64

Rezultatul va fi d29yZHByZXNzdXNlcg==, care este valoarea ce trebuie să o includeți în fișierul YAML.

După ce am preparat fișierul YAML, aplicați-l folosind kubectl:

kubectl apply -f mariadb-user-secret.yaml

În exemplele de resurse Kubernetes care necesită acces la baza de date (cum ar fi Deployment-ul pentru aplicația WordPress sau StatefulSet pentru MariaDB Galera), vom referi acest Secret pentru a injecta credențialele în containere. De exemplu, în Deployment pentru WordPress, putem utiliza variabile de mediu pentru a specifica numele de utilizator și parola:

env:

- name: WORDPRESS\_DB\_USER

valueFrom:

secretKeyRef:

name: mariadb-user-secret

key: username

- name: WORDPRESS\_DB\_PASSWORD

valueFrom:

secretKeyRef:

name: mariadb-user-secret

key: password

Aceasta asigură că aplicația noastra poate accesa baza de date folosind credențialele stocate în mod sigur, fără a expune informații sensibile.

**Configurarea Kopf operator ce orchestreaza si pastreaza starea serviciilor:**

Implementarea unui operator Kubernetes cu KOPF pentru a gestiona deployment-ul unei imagini WordPress ce utilizează MariaDB Galera pentru stocare implică crearea unui script Python care să definească logica operatorului. Operatorul va asculta evenimente specifice legate de resursele Kubernetes definite de dumneavoastră (cum ar fi crearea, actualizarea sau ștergerea unei instanțe Custom Resource (CR) pentru deployment-ul WordPress) și va executa operațiunile necesare pentru a gestiona ciclul de viață al acestor instanțe conform specificațiilor.

**Pasul 1: Instalam KOPF**

Pentru a începe, trebuie să instalați KOPF și clienții Kubernetes Python:

pip install kopf Kubernetes

**Pasul 2: Definim Logica Operatorului**

Crearea fișierului wordpress\_operator.py va conține logica operatorului. Operatorul va răspunde la evenimente pentru a crea resursele necesare pentru a rula WordPress și MariaDB Galera.

import kopf

from kubernetes import client, config

# Configurarea clientului Kubernetes

config.load\_kube\_config()

@kopf.on.create('mycompany.com', 'v1', 'wordpresssites')

def create\_fn(spec, \*\*kwargs):

name = kwargs.get('name')

namespace = kwargs.get('namespace', 'default')

# Date pentru MariaDB

db\_user = spec.get('dbUser')

db\_password = spec.get('dbPassword')

db\_name = spec.get('dbName')

# Date pentru WordPress

wp\_image = spec.get('wpImage', 'wordpress:latest')

# Crearea Secret pentru baza de date

secret\_data = {

"username": base64.b64encode(db\_user.encode()).decode(),

"password": base64.b64encode(db\_password.encode()).decode(),

}

secret = client.V1Secret(

metadata=client.V1ObjectMeta(name=f"{name}-db-secret"),

data=secret\_data,

type="Opaque",

)

# Crearea StatefulSet pentru MariaDB Galera

create\_mariadb\_statefulset(namespace, name, db\_user, db\_password, db\_name)

# Crearea Deployment pentru WordPress

create\_wordpress\_deployment(namespace, name, wp\_image)

client.CoreV1Api().create\_namespaced\_secret(namespace, secret)

# Logica pentru crearea celorlalte resurse (PVC, Deployment, Service, etc.)

def create\_mariadb\_statefulset(namespace, name, db\_user, db\_password, db\_name):

statefulset = V1StatefulSet(

metadata=V1ObjectMeta(

name=f"{name}-mariadb",

namespace=namespace

),

spec=V1StatefulSetSpec(

serviceName=f"{name}-mariadb",

replicas=3,

selector={'matchLabels': {'app': f"{name}-mariadb"}},

template=V1PodTemplateSpec(

metadata=V1ObjectMeta(labels={'app': f"{name}-mariadb"}),

spec={

'containers': [

V1Container(

name="mariadb",

image="mariadb:10.5",

ports=[V1ContainerPort(container\_port=3306)],

env=[

V1EnvVar(name="MYSQL\_ROOT\_PASSWORD", value=db\_password),

V1EnvVar(name="MYSQL\_DATABASE", value=db\_name),

V1EnvVar(name="MYSQL\_USER", value=db\_user),

V1EnvVar(name="MYSQL\_PASSWORD", value=db\_password)

],

volumeMounts=[V1VolumeMount(mount\_path="/var/lib/mysql", name="data")]

)

]

}

),

volumeClaimTemplates=[

V1PersistentVolumeClaim(

metadata=V1ObjectMeta(name="data"),

spec=V1PersistentVolumeClaimSpec(

accessModes=["ReadWriteOnce"],

resources=V1ResourceRequirements(

requests={"storage": "10Gi"}

)

)

)

]

)

)

def create\_wordpress\_deployment(namespace, name, wp\_image, db\_name, db\_user\_secret\_name):

deployment = V1Deployment(

metadata=V1ObjectMeta(name=f"{name}-wordpress", namespace=namespace),

spec=V1DeploymentSpec(

replicas=2,

selector=V1LabelSelector(matchLabels={"app": f"{name}-wordpress"}),

template=V1PodTemplateSpec(

metadata=V1ObjectMeta(labels={"app": f"{name}-wordpress"}),

spec={

'containers': [

V1Container(

name="wordpress",

image=wp\_image,

ports=[V1ContainerPort(container\_port=80)],

env=[

V1EnvVar(

name="WORDPRESS\_DB\_HOST",

value="mariadb-service:3306"

),

V1EnvVar(

name="WORDPRESS\_DB\_NAME",

value=db\_name

),

V1EnvVar(

name="WORDPRESS\_DB\_USER",

valueFrom=V1EnvVarSource(

secretKeyRef=V1SecretKeySelector(

name=db\_user\_secret\_name,

key="username"

)

)

),

V1EnvVar(

name="WORDPRESS\_DB\_PASSWORD",

valueFrom=V1EnvVarSource(

secretKeyRef=V1SecretKeySelector(

name=db\_user\_secret\_name,

key="password"

)

)

),

]

)

]

}

)

)

)

client.AppsV1Api().create\_namespaced\_deployment(namespace=namespace, body=deployment)

return {'message': f"WordPress site '{name}' created with MariaDB Galera!"}

Acest exemplu de bază arată cum să răspundeți la evenimente de creare pentru resurse custom, cum ar fi wordpresssites sub grupul mycompany.com. Când o astfel de resursă este creată, operatorul va iniția crearea unui secret Kubernetes pentru stocarea credențialelor bazei de date și va continua cu restul resurselor necesare.

**Pasul 3: Rulam Operatorul**

Pentru a rula operatorul, executați scriptul folosind KOPF:

kopf run wordpress\_operator.py

KOPF va începe să asculte evenimentele Kubernetes pentru resursele specificate și va executa funcțiile definite atunci când sunt declanșate evenimentele corespunzătoare.

**Bibliografie:**

<https://en.wikipedia.org/wiki/Kubernetes>

<https://kubernetes.io/docs/concepts/extend-kubernetes/operator>

<https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources>

<https://mariadb.com/kb/en/what-is-mariadb-galera-cluster>

<https://severalnines.com/blog/running-galera-cluster-kubernetes>

<https://www.bmc.com/blogs/kubernetes-crd-custom-resource-definitions>

https://webthesis.biblio.polito.it/22714/1/tesi.pdf