Scavenger jobs operator

Cieľ

- vyrobiť Custom Resource Definition (CRD) *ScavengerJob* a k nemu controller, ktoré budú spolu slúžiť na zvyšovanie využitia zdrojov clustru pomocou Operator Pattern (https://kubernetes.io/docs/concepts/extend-kubernetes/operator/)
- black-box prístup - operátor nerieši, čo beží
- vytvoriť prototyp aplikácie, ktorá sa dá spustiť cez *ScavngerJob* (GROMACS, AMBER, PYTORCH/TENSORFLOW - tu nejaký malý mám)
- ide o prototypovéno **plne funkčné** riešenie, počíta sa s ďalším vývojom a úpravami

Technológie:

- https://tag-app-delivery.cncf.io/whitepapers/operator/
- odporúčam použiť Operator SDK https://operatorframework.io pretože vygeneruje celú kostru
- intro do Operator pattern od IBM https://developer.ibm.com/articles/introduction-to-kubernetes-operators/ , krátky tutorial na memcached operátor https://developer.ibm.com/learningpaths/kubernetes-operators/develop-deploy-simple-operator/create-operator/ (fajn si prejsť na pochopenie tvorenia CRD Spec, Status a controller reconcile() loop)
- https://cloud.redhat.com/blog/kubernetes-operators-best-practices
- https://github.com/operator-framework/operator-sdk/issues/6035 … ako spracovávať vŠetky CRD naraz
- https://omerxx.com/k8s-controllers/ good to know
- kľudne aj iné zdroje
- potrebné si naštudovať Priory Classes a pod preemption: https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/

Motivácia:

Cloudové prostredia sú notoricky známe tým, že trpia nízkou vyťaženosťou oproti rezerváciam. Scavenger Job (ďalej už len SJ) je job, ktorý slúži na zvýšenie utilizácie zdrojov clustru tým, že konzumuje voľné zdroje (ktoré ale môžu byť rezervované, no sú aktuálne nepoužívané), no ak príde do clustru požiadavka na použitie týchto rezervovaných zdrojov od “vlastníka” rezervácie, (pre nás abstrakcia “akýkoľvek workload, ktorý má vyššiu prioritu než SJ”), tak je SJ prerušený, uvoľní zdroje a tie si vezme workload s vyššou prioritou. Všetky SJ majú nižšiu prioritu než všetky iné druhy workloadov v clustri.
SJ môžu byť aj multipodove = jeden SJ môže chcieť spustiť Joby vo viacerých rôznych podoch a potencialne viacerych nodoch zároveň. Ak je takýto SJ prerušený, všetky jeho Joby musia byť zrušené.

SJ pomáhajú s využítím clustru a neplytvaním zdrojmi no aj samotne SJ môžu trpieť niektorými týmito problémami :

1. môžu plytvať zdroje, ak SJ by teoreticky dobehol o 3 minúty a už beží 2 hodiny a my ho ukončíme -> zbytočné prepočítaný CPU čas, úloha sa musí počítať znova
2. môžu plytvať zdroje ak si SJ požiadajú o zdroje, ktoré nevyužijú z väčšej časti -> rovnaký problém nevyužitia zdrojov

Návrh CRD:

CRD *ScavengerJob*:

ScavengerJob bude iba abstrakcia, reálne bude kontrolerom spúšťaný klasicky *kind: Job*, ktorý preberie informácie zo specu SJ (a Job spúšťa samozrejme Pod). Job spustený jedným SJ musí mať ownerRef na SJ, prípadne inú jasnú viazanosť na to, ku ktorému SJ patrí.

Spec musí obsahovať:

3. image, ktorý bude použitý jobom
4. požiadavky na zdroje (pripraviť na requests i limits, používať sa naozaj budú guarabteed pody, ktoré majú req==lim)
5. referenciu na PVC, CM, Secrets ktoré sa majú použiť atď (môže ich byť aj viac, može to byť štrukturované ako pre normálne Joby čiže meno objektu a mountPath kde ho pripojiť)
6. command na spustenie

7. či ide o multikontajnerovú úlohu, ak áno koľko kontajnerov (resourcz su rovnaké pre každý kontajner) (melo by vyplnit spise MPIJob kind)
8. runAsUser
9. ďalšie sa možno vyskytnú

Status musí obsahovať:

10. informáciu o stave SJ v rámci clustru: čaká, running, prerušený, completed, failing
11. informáciu o tom, koľko krát bol prerušený *interruptedCount*
12. creation Timestamp (to je možné, že je by default vygenerované v rámci CRD)
13. interruptedTimestamp: možno pole? značiace kedy bol SJ prerušený?
14. runningTimestamp: možno pole? značiace kedy bol SJ začatý (pole preto bo može byť prerušený a potom musí začať znova dakedy)?
15. finishedTimestamp (to je možné, že je by default vygenerované v rámci CRD)
16. priorityStartOverTimestamp: značí, kedy dobehlo posledných duration/*interruptionAfter* minutes (kvoli jednoduchšiemu počítaniu + 30m)
17. voliteľný message keby náhodou

Tieto informácie by mohli byť i v nejakej DB z ktorej by kontroler čítal, no riešenie chceme cloud-native (meaning?). Naviac, použitie CRD je vhodné aj vzhľadom na to, že tieto objekty môžu byť potom uvažované ako ďalší aktéri clustru či použité do metrík rovno z clustru.

Aby sme vyriešili bod 1. s plytvaním zdrojov:

18. implementujeme hodnotu *duration (interruptionAfter, ap.)*, ktorá značí dobu, ktorú workload MUSÍ BEŽAŤ aby bol považovaný za vhodný scavenger job.
19. požadujeme od workloadov, ktoré chcú bežat ako SJ aby implementovali checkpointy (v nejakej forme) a aby po reštarte tej istej úlohy neskôr sa vedel workload reštartovať od daného checkpointu. Tieto checkpointy by sa mali diať každých x<*duration* aby sa zaručilo, že daný čas nebol preppočítaný zbytočne a prípadne ukončený workload môže pokračovať
20. S plynutím tejto doby sa zvyšuje “dopočítanosť” workloadu a teda je menej a menej vhodný na ukončenie. Keďže môže bežať viac SJ zároveň a môžu byť v rôznej fázi rozpočítanosti, je potrebné aby bežiace SJ mali medzi sebou tiež nejaké priroty aby kube-scheduler (default plánovač v K8s) vybral v prípade potreby ten SJ, ktorý je zatiaľ v najnižšej fáze rozpočítanosti
21. Priorita “running SJ” (vzhľadom na jeho zrušenie, čím vyššia priorita, tým nižšia šanca na preempciu daného SJ) sa zvyšuje proporčne k *duration*, ak je *duration=30min*, tak môžme túto dobu rozkúskovať s rôznou granularitou, ukáže sa čo je efektívne (môže záležať aj od typu workloadu, uvidíme časom) napr:
    a. 3 *runningPriority*(zvýšenie po 10m)
    b. 5 *runningPriority* (zvýšenie po 6 minútach)
    c. atď.
    d. Toto zvyšovanie priority zabezpečuju reconcile loop operatoru keďže tá je zodpovedá za pravidelné kontrolovanie CRD. Dá sa skontrolovať čas vytvorenia SJ a potom po X minútach zvyšovať túto hodnotu
22. Ak je SJ prerušený alebo zrušený
    a. Ak končí Job (a teda Pod pod ním) s exit 1 a pod.status.phase-=Failed, SJ sa už nebude považovať ďalej za vhodný pretože v ňom očividne niečo nefunguje a nastaví sa celý SJ ako Failed alebo čosi také
    b. ak bol Pod prerušený z von (čiže na pod s pod.status.phase=Running sa aplikovala operácia DELETE) tak sa proste zrušia jeho Job(y) no SJ neprestane existovať, je iba označený ako prerušený a navýší sa mu *interruptedCount*
        i. *interruptedCount* určuje prioritu medzi čakajúcimi SJ. Ak sa znova uvoľní miesto a nejaký SJ môže bežať, najprv sa spustia (ak je miesto) SJ ktoré majú najvyšší *interruptedCount*. (Prístup “better finish what we started than start something else”). Ak nie je miesto na žiadny z už začatých a prerušených SJ, spustí sa nejaký čo ešte nebežal a je na nenho dostatok zdrojov

Aby sme vyriešili bod 2. s plytvaním zdrojov:
Dajú sa použiť tunery alebo ML techniky nastavenia vhodných zdrojov pre SJ. To nie je zatiaľ scope of this work.

Cca návrh controlleru (reconcile loop):

23. Reconcile by ~~snáď mohlo vrátiť nie 1 SJ per time instant ale celý list všetkých objektov typu *ScavengerJob* aby sa z nich dalo vyberať, ktorý bude vykonaný~~ ~~by mal byť triggered na základe zmeny Podu~~ mal byť triggered na
    a. Základe updatu z Jobu owned by some SJ (ktorý bol spôsobený zmenou Podu)
    b. Periodicky ak je SJ vo fáze Running a teda je potrebné mu zvyšovať running prioritu

24. Keď sa vytvorí novy SJ tak
    a. SJ na spustenie sa vyberajú podľa tej popísanej priority *interruptedCount* . Vybere sa ten, s najvyššou prioritou. Reconcile loop dostane po zaznamenaníCREATE v clustri ctrl.request vktorom je namespace ktorý spustil reconcile a objekt ktorý ho spustil. Ak ide o nový SJ objekt, je nutné (ASI?) prejsť working queue a nájsť,či neexistuje iný SJ s vyššou prioritou. Ak žiadny nie je, tak sa môže pustiť tento SJ čo teraz prišiel = vyrobí sa Job object a pustí.
        i. treba vytvoriť Job object, pripojiť všetky PVC, Secrets, ConfigMaps atď. Skontrolovať že existujú, su validne hodnoty atd Ak nie, nastaviť SJ na Failed so správou prečo a neskúšať ďalej, ak chyba je napr v neexistujúcom PVC apod.
        ii. Dá sa vytvoriť customPopFromWorkQueue, dotaz na ChatGPT "***implement custom logic of popping from work queue"*** , predtym uz mal kontext ohladom CRD a operatorov

In this example:

- The **popFromWorkQueue** function is where you implement your custom logic for dequeuing an item from the work queue. You can use **r.WorkQueue.Get()** to get an item from the work queue.
- The **enqueueToWorkQueue** function is used to add an item back to the work queue if requeuing is needed.

Note: The example uses **ctrl.Request** as the item type for simplicity. You might need to adapt the code based on the actual item type used in your work queue.

Remember to ensure that the work queue is properly closed when the controller is shutting down to avoid leaks. The **stopCh** channel in the **Run** function is commonly used to signal the controller to stop its reconciliation loop.

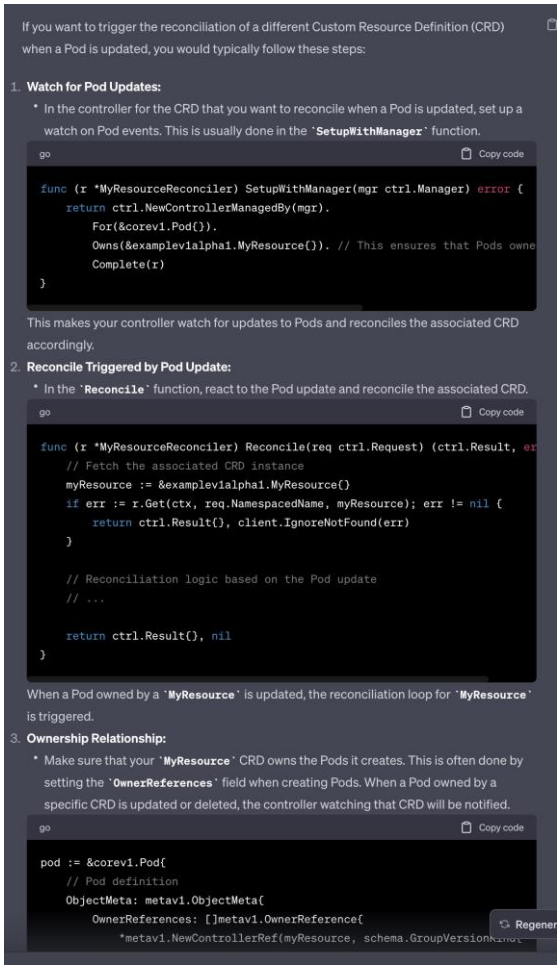
```
import (
"time"

ctrl "sigs.k8s.io/controller-runtime"
"sigs.k8s.io/controller-runtime/pkg/client"
)

func (r *MyResourceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
// Fetch the resource using the request's NamespacedName
myResource := &examplev1alpha1.MyResource{}
if err := r.Get(ctx, req.NamespacedName, myResource); err != nil {
return ctrl.Result{}, client.IgnoreNotFound(err)
}

// Your custom logic here...

return ctrl.Result{}, nil
}

func (r *MyResourceReconciler) Run(stopCh <-chan struct{}) {
for {
select {
case <-stopCh:
// Stop the reconciliation loop if the stop signal is received
return
default:
// Continue processing
}

// Implement your custom logic for popping from the work queue
key, quit := r.popFromWorkQueue()

if quit {
// Stop processing if the work queue is closed
return
}
```

```
// Convert the key back to a request object
req := reconcile.Request{NamespacedName: key.(client.ObjectKey)}

// Process the resource
result, err := r.Reconcile(req)
if err != nil {
// Handle error
}

// Requeue the resource if needed
if result.Requeue {
r.enqueueToWorkQueue(req)
}

// Wait for a short duration before processing the next item
time.Sleep(time.Second)
}
}

// popFromWorkQueue implements custom logic for dequeuing an item from the work queue
func (r *MyResourceReconciler) popFromWorkQueue() (interface{}, bool) {
// Implement your custom logic for dequeuing
// This might involve interacting with the work queue and returning the popped item

// For example, using ctrl.Request as the item type
key, quit := r.WorkQueue.Get()

return key, quit
}

// enqueueToWorkQueue adds an item to the work queue
func (r *MyResourceReconciler) enqueueToWorkQueue(req reconcile.Request) {
r.WorkQueue.Add(req.NamespacedName)
}
```

b. Job sa tupo vytvorí a to je všetko. Čokoľvek čo sa ďalej stane s Jobom triggerne reconcile (malo by) , napr.
   i. ak Job ostane pending kvôli nedostatku zdrojov spôsobí to reconcile kde sa zistí dôvod (nedostatok zdrojov) a SJ ostane v stave Pending. Reconcile na taký SJ napr s return ctrl.Result(RequeueAfter: time), nil Zároveň třeba zmazať vytvorený Job pretože bude visieť v systéme a čakať na zdroje kým nebudú dostupné a to nechceme. SJ sa nezmaže, ten ostane Pending a keď na neho znova príde rad, spustí sa Job, ktorý potom triggerne reconcile atď
      1. Otázka - má mať SJ, na ktorý neboli zdroje časom vyššiu prioritu? Prevent starving. Alebo je to nám jedno a má si to riešiť ten, kto ich submitol že jako dlho kvasia v queue a že možno žiadajú príliš veľa zdrojov. Alebo ich my requeue náhodne po nejakom čase a možno budú vtedy potrebné zdroje dostupné? ... -> **do začiatku stačí že sa takýto Job reconcile(requeuAfter: nejaký čas)**
   ii. Ak Job zfailuje, nastaviť SJ jako failed a už na ňom neoribť žiadne akcie (check v controlleri ci SJ.status == failed a ak hej rovno return)
   iii. Ak Job Succeded, podobne jak fail, nič s nim už nerobiť
   iv. Ak Job Running, tak pre SJ platí že sa im musí cyklicky meniť *runningPriority* - narastať do 30m od behu, potom nastaviť priorityStartOverTimestamp, nastaviť prioritu zas na najnižšiu a ideme dokola

c. Funkcie na zaregistrovanie eventu pre pody
   i. SetupWithManager to watch for Job/Pod updates .. kedze pod je spusteny jobom a job moze mat owner reference na SJ .. Job.. ScavengerJob
   ii. EnqueueRequestsFromMapFunc https://pkg.go.dev/sigs.k8s.io/controller-runtime/pkg/handler#EnqueueRequestsFromMapFunc viac komplikované predikáty
   iii. https://sdk.operatorframework.io/docs/building-operators/golang/references/event-filtering/ možno budeme potrebovať toto

d. AKO MNOŽSTVO VOĽNÝCH ZDROJOV SA BERIE DO ÚVAHY THRESHOLD=70% CELKOVO REÁLNYCH ZDROJOV CLUSTRU ABY OSTALO NEJAKÉ VOĽNÉ MIESTO DO ISTOTY.
   i. threshold sa dá tunovať podľa výsledkov, heuristikou sa možno príde na vhodnú hdnotu pre nás a naše zloženie workloadov (a im podobné setupy)
   ii. otázka je, že či sa má počítať aktuálny usage a z toho 70% alebo aktuálny usage plus SJ ktorý sa uvažuje a z toho 70%



If you want to trigger the reconciliation of a different Custom Resource Definition (CRD) when a Pod is updated, you would typically follow these steps:

1. **Watch for Pod Updates:**
   * In the controller for the CRD that you want to reconcile when a Pod is updated, set up a watch on Pod events. This is usually done in the `SetupWithManager` function.

```go
func (r *MyResourceReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&corev1.Pod{}).
        Owns(&examplev1alpha1.MyResource{}). // This ensures that Pods owne
        Complete(r)
}
```

   This makes your controller watch for updates to Pods and reconciles the associated CRD accordingly.
2. **Reconcile Triggered by Pod Update:**
   * In the `Reconcile` function, react to the Pod update and reconcile the associated CRD.

```go
func (r *MyResourceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, er
    // Fetch the associated CRD instance
    myResource := &examplev1alpha1.MyResource{}
    if err := r.Get(ctx, req.NamespacedName, myResource); err != nil {
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }

    // Reconciliation logic based on the Pod update
    // ...

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

   When a Pod owned by a `MyResource` is updated, the reconciliation loop for `MyResource` is triggered.
3. **Ownership Relationship:**
   * Make sure that your `MyResource` CRD owns the Pods it creates. This is often done by setting the `OwnerReferences` field when creating Pods. When a Pod owned by a specific CRD is updated or deleted, the controller watching that CRD will be notified.

```go
pod := &corev1.Pod{
    // Pod definition
    ObjectMeta: metav1.ObjectMeta{
        OwnerReferences: []metav1.OwnerReference{
            *metav1.NewControllerRef(myResource, schema.GroupVersionKind{
```

Poznámky:
Od bodu existencie SJ v clustri musia mať VŠETKY workloady prioritu napr *ordinaryWorkloadPriority*. Vytvoriť webhook ktorý nastaví dostatočne vysoký default všetkým ostatným (dostatočne vysoký = aby ostalo dosť čísel reprezentujúcich priority SJ)
ordinaryWorkloadPriority je non-preemptible
scavengerWorkloadPriority je preemptible

Ak sa má ScavengerJob skontrolovať na validitu, musí byť pred ním ValidatingAdmissionConfiguration/Webhook. Po prijatí do clustru už nie je možne odstrániť objekt.

Demo Aplikácia

Generická demo aplikácia

vytvoriť nejaký SJ ktorý dostatočne dlho trvá a tvorí checkpointy. je jedno čo, nech to otestuje že funguje framework

GROMACS

vytvoriť DB GromacsToTune ktoré budú obsahovať informácie o GROMACS výpočte, ktorý ďalej poputuje do tunera. Tuner snáď vypočíta vhodnú resource konfiguráciu (req, lim) a s informáciami GromacsToTune (a možno ešte extra, tie ktoré treba do SJ) vyrobiť predpisy ScavengerJobs a submitnúť do clustru.
Submit môže byť do personal namespacu ale na *ScavengerJobs* sa nevzťahuje namespace ani project kvóta. (zatiaľ)
Tento gromacsScavengerJob framework si môže držať zoznam napočítaných konfigurácií pre SJ a ak sa nejaká ne a ne spustiť (napr nedostatok zdrojov), odstrániť ten SJ a skúsiť inú, menšiu.

Vylepšenia na neskôr (hlavná funkcionalita implementovaná a otestovaná):

- ak SJ neimplementuje checkpointy, dá sa uvažovať nad budúcou funkcionalitou K8s *snapshot* kontajneru - zapnúť do alfa verzie, pri interuption vykonať snapshot ——> ako funguje znovuspustenie neviem
- zatiaľ predpokladáme, že dáta sú už na PVC a nemusia tam byť nakopírované ani stianuté (žiadny input filler ap.) —— —> potom možno nepotrebné žiadať lokaciu dát v SJ Specu inú než kam pripojiť PVC?
- metrics endpoint - push metrics ktoré hlásia kedy boli SJ spustené, dokončené, prerušené, koľkokrát, po koľkých minútach od prijatia do clustru sa spustili atď.
- kvóty na ScavengerJobs - čo ak si niekto prerobí všetky workloady do SJ a tým zahltí cluster a budú sa počítať na extra prístupných zdrojoch len jeho workloady? tým by obišiel personal kvótu
- prejsť z blackbox prístupu na white box kde sa sleduje využitie zdrojov SJ a ak to neni moc dobré, nejak poupraviť (napr podľa historickej usage)
- optimalizácia thresholdu:
    - 70% nodu
    - balancovať medzi vačšimi a menšími nodmi

- MPIJob