TP Kubernetes

Consignes

Vous utiliserez la même VM de travail que pour le TP docker.

Evaluation

Durant le TP, vous devrez répondre à des questions et effectuer des actions (encadrés en couleur).

Elles sont identifiées et numérotées (ex: KQ1 - Kubernetes Question 1). Un encadré est numéroté et peut regrouper plusieurs questions.

Au début du TP, constituez-vous un fichier avec votre traitement de texte favori et remplissez le au fur et à mesure du TP en reprenant bien la référence de l'encadré (ex: KQ1), N'hésitez pas à inclure des captures d'écrans lorsque c'est pertinent ou même des schémas si vous voulez illustrer un choix d'architecture de votre part.

En fin de TP, vous devrez poster le résultat de votre travail au format PDF dans la section devoirs sur EDUNAO

Le fichier doit être nommé par <votre nom>-tpkube-<date>.pdf

Par exemple : claude-tpkube-20230525.pdf

Dans les questions (cadres en jaune) il est parfois indiqué BONUS : ces questions sont souvent assez/très difficiles et demandent du temps de recherche. N'hésitez pas à les sauter (vous y reviendrez plus tard s'il vous reste du temps), elles comptent peu pour l'évaluation, il s'agit plus d'un challenge ou d'une façon d'aller plus loin pour les étudiants déjà familiers avec le thème

Le cluster MicroK8S

C'est un cluster mono-machine (1 node).
Nota : Il est toutefois possible d'ajouter des nodes, et en fin de TP et si vous avez du temps et si vous disposez d'une machine puissante, vous pourrez ajouter un node en suivant ces instructions.
Il est possible d'étendre les fonctions du cluster par des add-ons . Plusieurs add-ons sont présents dans le cluster de la VM :

- Storage
- DNS
- Ingress
- registry

Le metrics-server est également activé pour collecter les mesures d'usage du cluster.

La CLI (Command Line interface) `kubectl` est disponible. C'est l'outil de base pour interagir avec l'API Kubernetes et permettre à la fois de collecter des informations mais aussi de déployer de nouvelles ressources (ce que nous allons voir par la suite).

Prise en main du cluster MicroK8S

Vérifiez la configuration du Cluster
`kubectl cluster-info`

Pour disposer de tous les détails du cluster
`kubectl cluster-info dump`

Si ces commandes n'affichent rien ou des erreurs, contactez le formateur pour régler le problème.

Afin d'interroger le cluster, vous devrez ajouter aux commandes `kubectl` le nom du namespace : `--namespace default ou plus rapidement -n default`

Dans ce TP nous utilisons l'espace de noms par défaut "`default`" ou `""` (vide). Vous pouvez à tout moment vérifier le namespace qui est utilisé si vous ne le spécifiez pas. Il est défini au niveau du context :

```
kubectl config get-contexts
CURRENT   NAME     CLUSTER   AUTHINFO   NAMESPACE
*         admin    netobs0   admin
```

Mais si une commande ne trouve pas la ressource, posez-vous la question s'il est nécessaire de spécifier l'espace de noms.

Un aide mémoire des commandes kubectl est disponibe :
https://kubernetes.io/fr/docs/reference/kubectl/cheatsheet/#visualisation-et-recherche-de-ressources

Créer et manipuler des pods avec Kubernetes (distinction des modes impératifs et déclaratifs)

Vous allez voir que beaucoup de commandes de la CLI kubernetes vont vous rappeler des actions similaires que vous avez effectuées avec docker, vous devriez rapidement trouver vos marques.

Créons (en mode impératif) la ressource deployment qui gère pour vous la disponibilité des pods (ensemble de containers). C'est la ressource la plus utilisée dans Kubernetes

```
$ kubectl create deployment mydeploy --image=nginx
deployment.apps/mydeploy created
```

Vous pouvez ensuite lister le deployment que vous avez créé (notez que la commande liste l'ensemble des deployment disponibles, si vous en aviez plusieurs, vous les verriez tous)

```
$ kubectl get deploy
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
mydeploy   1/1     1            1           10s
```

Vous pouvez bien sûr lister les replicasets et bien évidemment les pods (la plus petite unité dans Kubernetes)

```
$ kubectl get replicasets
NAME                   DESIRED   CURRENT   READY   AGE
mydeploy-7dff945675   1         1         1       3m57s

$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
mydeploy-7dff945675-jzqlz   1/1     Running   0          3m50s
```

Nous allons tout de suite voir une fonctionnalité très intéressante de Kubernetes, le scaling. Ajoutons 3 pods identiques à notre deployment (en plus de celui qui existe)

```
$ kubectl scale deployment mydeploy --replicas=4
deployment.apps/mydeploy scaled

$ kubectl get replicasets
NAME                   DESIRED   CURRENT   READY   AGE
mydeploy-7dff945675   4         4         4       9m37s
$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
mydeploy-7dff945675-f84k2   1/1     Running   0          27s
mydeploy-7dff945675-fj66b   1/1     Running   0          27s
mydeploy-7dff945675-jzqlz   1/1     Running   0          9m39s
mydeploy-7dff945675-zmnzb   1/1     Running   0          27s
```

Essayez de “killer” un des pods et observez ce qui se passe ensuite :

```
$ kubectl delete pod mydeploy-7dff945675-f84k2
pod "mydeploy-7dff945675-f84k2" deleted
$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
mydeploy-7dff945675-46f86   1/1     Running   0          7s
mydeploy-7dff945675-fj66b   1/1     Running   0          3m26s
mydeploy-7dff945675-jzqlz   1/1     Running   0          12m
mydeploy-7dff945675-zmnzb   1/1     Running   0          3m26s
```

Vous pouvez maintenant revenir à un seul replica de votre deployment (et faite très vite après un get pod à nouveau)

```
$ kubectl scale deployment mydeploy --replicas=1
deployment.apps/mydeploy scaled
$ kubectl get pods
NAME                         READY   STATUS        RESTARTS   AGE
mydeploy-7dff945675-46f86   0/1     Terminating   0          89s
mydeploy-7dff945675-fj66b   0/1     Terminating   0          4m48s
```

```
mydeploy-7dff945675-jzqlz   1/1     Running       0          14m
mydeploy-7dff945675-zmnzb   0/1     Terminating   0          4m48s
```

Vous pouvez facilement voir les logs d'un pod :

```
$ kubectl logs mydeploy-7dff945675-jzqlz
```

Dans kubernetes, une ressource events permet de voir un résumé de ce que fait l'orchestrateur, cela peut être très pratique quand un pod ne se déploie pas ou ne passe pas en running sans que les logs dans le container soient suffisants pour comprendre ce qui se passe (attention les events ne sont pas conservés plus de quelques minutes car il se passe beaucoup de choses sur un cluster) :

```
$ kubectl get events
$ kubectl get event --field-selector involvedObject.name=mydeploy
```

```
$ kubectl create deployment mydeploy2 --image=httpd
$ kubectl get events
15s         Normal   Scheduled
pod/mydeploy2-596d9987b7-rv7ff    Successfully assigned
default/mydeploy2-596d9987b7-rv7ff to minikube
14s         Normal   Pulling
pod/mydeploy2-596d9987b7-rv7ff    Pulling image "httpd"
11s         Normal   Pulled
pod/mydeploy2-596d9987b7-rv7ff    Successfully pulled image "httpd"
in 2.644452205s
11s         Normal   Created
pod/mydeploy2-596d9987b7-rv7ff    Created container httpd
10s         Normal   Started
pod/mydeploy2-596d9987b7-rv7ff    Started container httpd
15s         Normal   SuccessfulCreate
replicaset/mydeploy2-596d9987b7   Created pod:
mydeploy2-596d9987b7-rv7ff
15s         Normal   ScalingReplicaSet         deployment/mydeploy2
Scaled up replica set mydeploy2-596d9987b7 to 1
```

Tout comme pour docker, vous pouvez aussi vous connecter dans un pod ou lui demander d'exécuter des commandes

```
$ kubectl exec mydeploy-7dff945675-jzqlz -it -- bash
```

```
$ kubectl exec mydeploy-7dff945675-jzqlz -it -- cat /etc/passwd
```

Vous pouvez également obtenir un descriptif d'une ressource au format texte (mais non structuré) avec la commande describe

```
$ kubectl describe deployment mydeploy
```

Un des principes les plus puissants de Kubernetes est son fonctionnement en mode déclaratif. Cela signifie que plutôt que lui donner des ordres séquentiels, vous pouvez

décrire ce que vous souhaitez dans un fichier descripteur de ressources et lui demander de l'appliquer, il va se débrouiller tout seul pour réconcilier l'état actuel avec ce que vous avez déclaré dans votre manifest (descripteur)

Très intéressant, vous pouvez extraire depuis Kubernetes le descripteur d'une ressource si vous ne l'avez pas créé vous mêmes, vous pourrez toujours le récupérer pour ensuite le modifier. Il faut utiliser -o (output) et un format, on utilisera le plus souvent yaml (mais on peut utiliser aussi json, à noter le format wide qui affichera normalement mais avec plus d'informations qu'un get standard)

```
$ kubectl get deployment mydeploy -oyaml
```

Vous pouvez bien sûr rediriger ces éléments directement dans un fichier *(Attention, si vous utilisez le terminal intégré à vscode, la commande ci-dessous peut mal fonctionner, problème de compatibilité avec snap. Si vous avez un problème de redirection, vous pouvez taper la même commande dans un terminal en dehors de vscode)*

```
$ kubectl get deployment mydeploy -oyaml > mydeploy.yaml
$ vi mydeploy.yaml
```

Modifier le nombre de replicas dans le fichier, vous pouvez ensuite tester

```
$ kubectl diff -f mydeploy.yaml
$ kubectl apply -f mydeploy.yaml
```

En production, on utilise toujours des fichiers descripteurs qui sont gérés en configuration dans un repository git.
De plus vous avez sans doute remarqué que le descripteur extrait de Kubernetes contient beaucoup d'infos ajoutées par Kubernetes lui-même (comme des UID ou des infos sur le statut). Cela ne pose pas forcément de problèmes, mais il est préférable d'avoir une description la plus minimaliste possible dans un git, nous verrons un exemple à utiliser dans les exercices suivants.
Et bien sûr, très pratique pour des tests où l'on veut construire en repartant de zéro :

```
$ kubectl delete -f mydeploy.yaml
```

KQ1:

Observez-vous une relation particulière dans le nommage des ressources deployment, replicaset et pods ?

A noter que vous ne pouvez pas vraiment stopper un pod (comme avec docker stop ou pause), il n'y a pas non plus de commande stop pour un deployment. Que pouvez-vous imaginer pour arrêter les instances/replicas d'un deployment sans pour autant supprimer le deployment ?

BONUS : à votre avis, quelle peut-être la limitation du apply/delete et du mode déclaratif ? Si par exemple on souhaite renommer notre deployment ?

Accéder aux applications et containers déployés sur Kubernetes depuis l'intérieur et l'extérieur du cluster

INFO Pour la suite des exercices, nous présenterons des fichiers exemples, ils sont directement disponibles dans la VM du TP sous
`$HOME/tp-cs-containers-student/kubernetes`

Vous pouvez les observer et les déployer au fur et à mesure des étapes du TP en vous positionnant dans le répertoire :
`$HOME/tp-cs-containers-student/kubernetes`
puis
`kubectl apply -f <nom-du-fichier.yml>`

Nous allons explorer plusieurs manières d'accéder aux applications qui tournent dans les PODs, pour cela nous allons utiliser un descripteur de deployment simple qui expose une application web minimaliste sur le port 8080 du pod, voici la trame (fichier bgd.deploy.yml)

Les fichiers décrits sont également disponibles dans le répertoire $HOME/tp-cs-containers-student/kubernetes afin d'éviter les problèmes de copier/coller et d'indentation yaml. N'hésitez pas à les utiliser

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: bgd
spec:
 replicas: 1
 selector:
   matchLabels:
     app: bgd
 template:
   metadata:
     labels:
       app: bgd
   spec:
     containers:
     - image: quay.io/redhatworkshops/bgd:latest
       name: bgd
```

Déployez donc ce nouveau deployment
`kubectl apply -f bgd.deploy.yml`

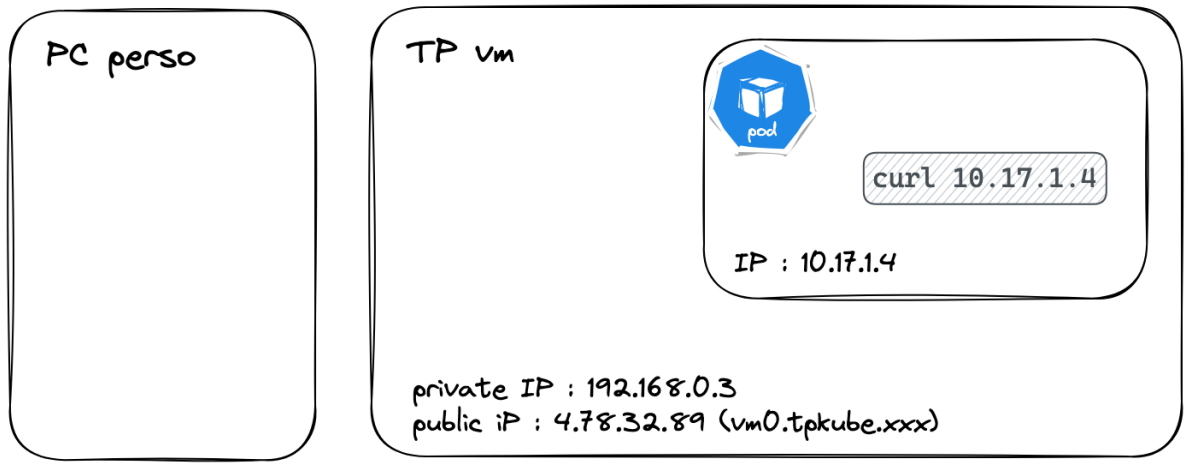
Tout d'abord, nous allons récupérer l'IP du pod, il y a plusieurs façon de le faire :

```
$ kubectl get pod -owide
$ kubectl get pod -l app=bgd -o custom-columns=IP:.status.podIP
$ kubectl get pod -l app=bgd -ojson | jq -r
'.items[].status.podIPs[].ip'
$ kubectl get pod -l app=bgd -o jsonpath="{.items[*].status.podIP}"
```

Utilisons la dernière version pour stocker l'IP dans une variable d'environnement

```
POD_IP=$(kubectl get pod -l app=bgd -o
jsonpath="{.items[*].status.podIP}")
```

Depuis le pod lui-même



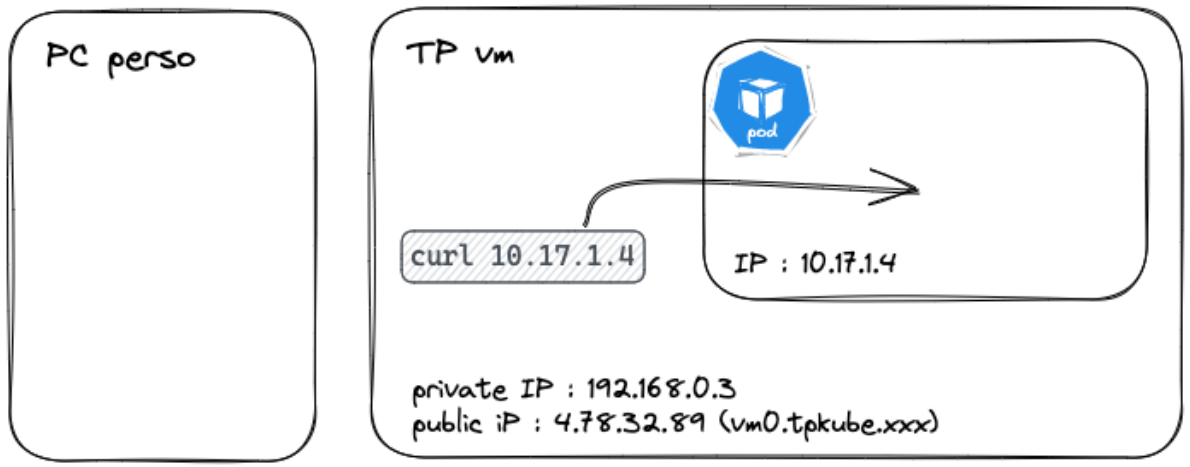
Appelons depuis l'intérieur du pod (via exec) le service pour voir s'il fonctionne et répond sur l'IP et port 8080 :

```
POD_NAME=$(kubectl get pod -l app=bgd \
      -o jsonpath="{.items[*].metadata.name}")

kubectl exec -it ${POD_NAME} -- bash -c \
      "curl -s $POD_IP:8080 |grep Welcome"
```

Vous pouvez changer le port pour vérifier que ça ne fonctionne pas, vous pouvez aussi tester sur localhost ou 127.0.0.1

Depuis le host directement vers le pod



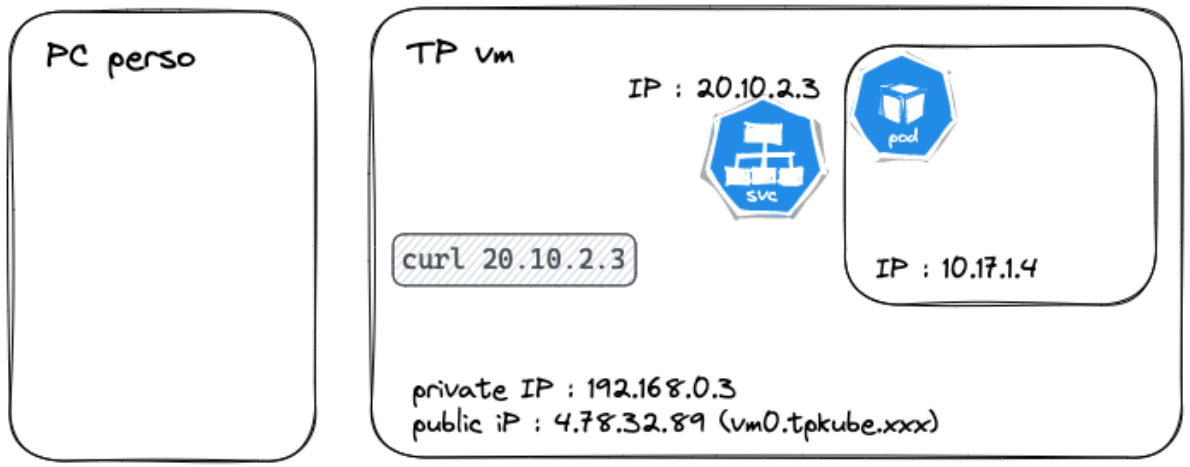
Un peu comme pour Docker, lorsque nous sommes sur un host du cluster Kubernetes (ce qui est le cas de notre VM de TP), nous avons accès au réseau des pods qui est rendu accessible via des règles IPtables (pour les curieux, plus de détails ici : https://github.com/canonical/microk8s/issues/72)

Vous pouvez donc accéder à l'application qui tourne dans un pod depuis le host en utilisant son adresse IP

```
POD_IP=$(kubectl get pod -l app=bgd \
      -o jsonpath="{.items[*].status.podIP}")
curl -s $POD_IP:8080 |grep Welcome
```

Comme sur Docker, on verra qu'il n'est pas possible d'utiliser le nom des pods (en revanche un service DNS existe pour les appels de pods entre eux mais nous allons voir cela un peu plus loin).

Utilisation d'un service type clusterIP



Sur Kubernetes, la façon d'exposer un pod au sein du cluster est de créer un service, voici le code d'un service que vous pouvez déployer pour exposer votre deployment
Notez qu'il s'agit ici du type ClusterIP qui va affecter au service une adresse IP sur le réseau des services du cluster Kubernetes.

fichier bgd.svc.yml

```yaml
apiVersion: v1
kind: Service
metadata:
 name: bgd
spec:
 ports:
 - port: 8080
   protocol: TCP
   targetPort: 8080
 selector:
   app: bgd
 type: ClusterIP
```

Pour information, on peut aussi le créer via une commande impérative (encore une fois, plutôt réservé à des tests, on ne fera jamais cela en production où l'on utilisera des services définis dans des fichiers yaml dans un repository de code)

```
kubectl expose deployment bgd --port 8080
```

```
kubectl apply -f bgd.svc.yml
```

à noter, cela dépend bien sûr de la distribution de Kubernetes et du SDN que vous utilisez mais sur microk8s (qui utilise CALICO comme SDN), le range du network des pods et des services est le suivant :
https://microk8s.io/docs/configure-cni
The default CIDR for pods is `10.1.0.0/16`. All pods are assigned an IP address in that range.
The default service CIDR is `10.152.183.0/24`. `10.152.183.1` will typically be reserved for the Kubernetes API, and `10.152.183.10` will be used by CoreDNS.

Vérifions ce qui se passe quand on appelle l'IP du service

```
SVC_IP=$(kubectl get svc bgd -ojson | jq -r ".spec.clusterIP")
curl -s ${SVC_IP}:8080 |grep Welcome
         <h1>Welcome to the Blue/Green Application!</h1>
```

Cela fonctionne car la couche réseau (SDN Calico) de microk8s permet d'accéder au réseau des services du cluster Kubernetes depuis le host .

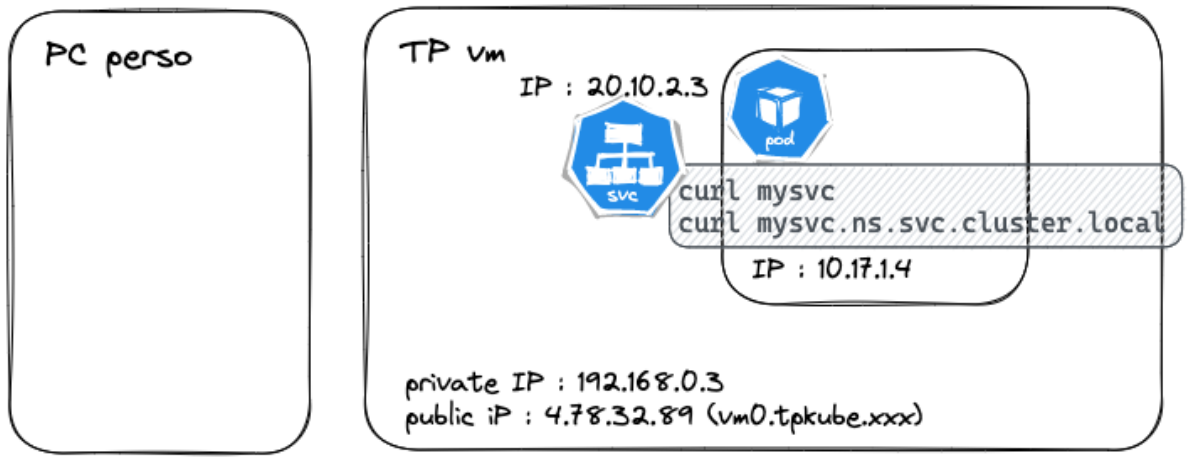
Cela est très pratique pour appeler des services au sein du cluster lui-même (*architecture micro services ou simplement plusieurs applications ou tiers d'une même application communiquant ensemble*)

On utilise un service car c'est lui qui va également réaliser le load balancing si on a plusieurs pods. Son adresse IP ne changera pas tant qu'il n'est pas supprimé.

Vous pouvez faire un scale du deployment pour avoir plusieurs pods et vérifier qu'à chaque appel, vous obtenez des réponses depuis un pod différent. L'application bgd affiche l'adresse IP du pod répondant à la requête dans sa réponse.

```
curl 10.152.183.219:8080 | grep -A 1 IP
                <td>Pod IP</td>
                <td>10.1.87.208</td>
```

Appel via les records DNS positionnés automatiquement avec les services



Comme nous l'avons vu, les services permettent d'exposer les pods au sein du cluster et réalise le loadBalancing en proposant une IP unique qui ne change pas tant que le service n'est pas supprimé.

Encore plus intéressant, Kubernetes propose un service DNS **interne** *(vous ne pouvez pas résoudre les noms Kubernetes depuis votre VM par exemple)* qui va créer automatiquement des records correspondant aux services (*et aussi aux pods mais les noms changeront régulièrement*)

Vous pouvez simuler l'appel d'un service depuis un pod en vous connectant dans un des pods de l'application déployée (bgd) et réaliser des requêtes sur le service qui expose votre application (*si vous le souhaitez, vous pouvez également déployer une autre application et faire les appels depuis ces pods*)

Les records DNS qui sont créés sont de type :

`<nom-du-service>.<nom-du-namespace>.svc.cluster.local`

Dans notre exemple, cela donne :
`bgd.default.svc.cluster.local`

De même le fichier /etc/resolv.conf définit des search domain permettant de compléter automatiquement le namespace courant (dans notre cas : default.svc.cluster.local)

Pensez à bien ajouter le numéro du port sur lequel l'application dans votre pod écoute.
Dans notre cas, une fois connecté dans le pod :
`curl bgd.default.svc.cluster.local:8080`
`curl bgd:8080`

KQ2:

Est-ce que les commandes fonctionnent ? Avec le nom long et le nom court ?

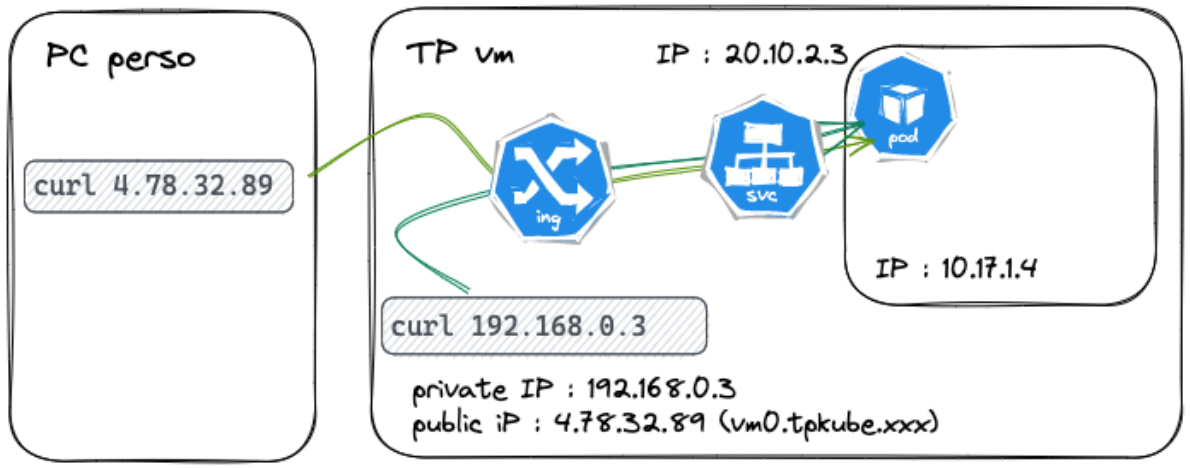
Pouvez-vous regarder le fichier /etc/resolv.conf pour comprendre pourquoi les noms courts fonctionnent ?

Essayer de supprimer le service (kubectl delete service bgd) et tenter de refaire les commandes curl, qu'obtenez-vous ?

Utilisation de service avec ingress controller

IMPORTANT

Cette solution doit être comprise car c'est celle que vous devrez utiliser ultérieurement dans le TP pour exposer l'application demoboard sur Kubernetes.



La solution idéale pour exposer vos services à l'extérieur du cluster est d'utiliser un ingress controller. Il s'agit d'un reverse proxy qui va recevoir les requêtes et les dispatcher ensuite vers les services kubernetes. Le gros intérêt est que l'ensemble de sa configuration va pouvoir être réalisé avec des descripteurs (donc depuis l'API de Kubernetes)

Avec la distribution microk8s, un ingress controller nginx est déjà installé, nous allons donc déployer une ingress rule pour le configurer afin de cibler notre service et l'exposer sur l'ingress controller

fichier bgd.ingress.yml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: bgd
spec:
 rules:
 - host: foo.bar.com
   http:
     paths:
     - pathType: Prefix
       path: "/"
       backend:
         service:
           name: bgd
           port:
             number: 8080
```

Attention, si vous aviez supprimer le service pour les test de records DNS précédemment, il faut le redéployer

```
kubectl apply -f bgd.svc.yml
kubectl apply -f bgd.ingress.yml
```

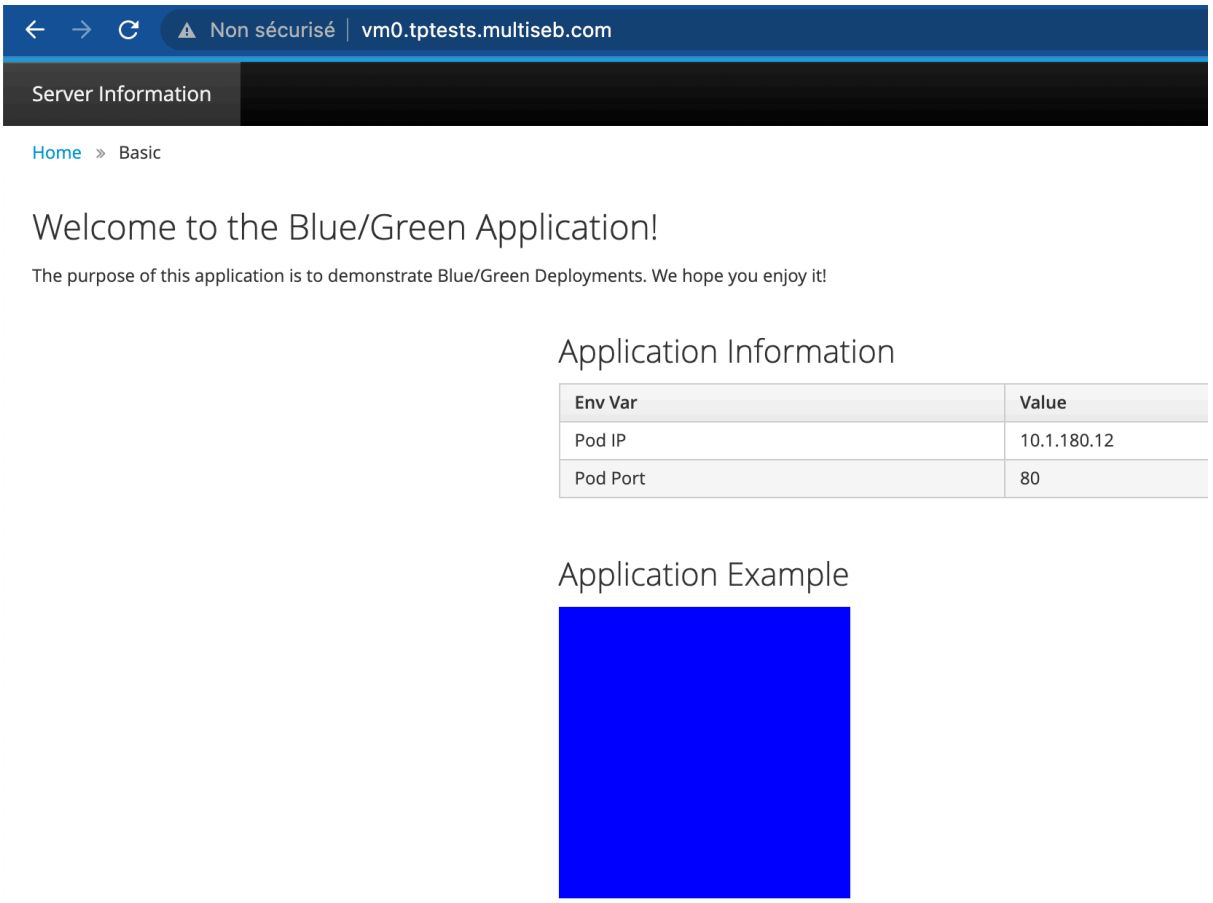
Pour information, l'ingress controller nginx de microk8s est un daemonset qui est configuré pour exposer les ports 80 et 443 en mode nodePort, ainsi un curl localhost sur le host ubuntu renvoie la page par défaut de l'ingress controller (404)

On va ensuite pouvoir appeler l'adresse IP de l'ingress controller (ici notre host ubuntu) et on va passer le Header http Host comme si on appelait le serveur foo.bar.com

```
curl -s -H "Host: foo.bar.com" localhost | grep Welcome
          <h1>Welcome to the Blue/Green Application!</h1>
```

Vos vms ont les ports 80 et 443 ouverts sur INTERNET largement (ce qui n'est pas la meilleure pratique de sécurité mais convient pour notre TP).
Vous pouvez donc vérifier que vous accédez depuis votre propre poste (et non plus depuis la VM ubuntu) à votre application.
Encore mieux, si vous changez l'entrée host dans le descripteur ingress avec le record dns associé à votre VM, vous pouvez accéder à l 'application depuis votre navigateur personnel à travers INTERNET.

```
- host: vmXX.tpcsonline.org
```



Utilisation de port-forward

BONUS

Si vous n'êtes pas en avance, vous pouvez sauter ce paragraphe.

Bon à savoir, plutôt dans le cadre d'une utilisation depuis un poste de développeur disposant d'un accès à kubectl, vous pouvez proxifier un port local vers un port d'un pod ou d'un service.

Ainsi, depuis une machine distante (comme votre PC personnel), si vous avez configuré kubectl (il faut installer le binaire sur votre poste et copier depuis la vmXX vers votre poste le fichier $HOME/.kube/config et l'éditer pour indiquer l'IP externe de la VM TP et mettre le TLS insecure, le port 16443 est ouvert sur INTERNET), vous pouvez réaliser les commandes suivantes :

```
$ kubectl --kubeconfig config port-forward deployments/bgd 1111:8080
Forwarding from 127.0.0.1:1111 -> 8080
Forwarding from [::1]:1111 -> 8080
```

Depuis un autre terminal

```
$ curl -s 127.0.0.1:1111 | grep Welcome
          <h1>Welcome to the Blue/Green Application!</h1>
```

Fermer le port-forwarding sur le premier terminal avec ctrl+c

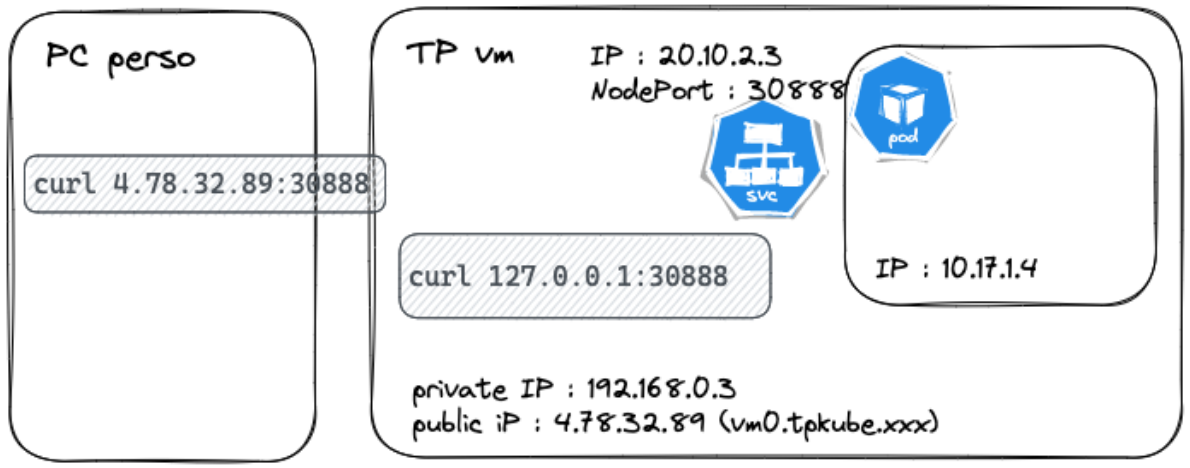
Cela peut vraiment être très pratique quand on a pas d'ingress controller et que l'on veut juste tester son application. Toute la communication va passer à travers l'API sans nécessiter d'ouvrir des ports supplémentaires.
C'est bien sûr totalement inutilisable en production puisque l'accès n'est possible que depuis le host qui a mis en place le proxy via kubectl port-forward (et si le process se termine, il ne sera pas relancé automatiquement)

Pour ceux qui veulent aller plus loin sur ce mode, la documentation de référence :
https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/
https://kubernetes.io/docs/concepts/cluster-administration/proxies/

Utilisation de services avec nodePort

BONUS

Si vous n'êtes pas en avance, vous pouvez sauter ce paragraphe.



Il est possible d'exposer le service pour des machines en dehors du cluster (par exemple votre PC personnel).

Une des solutions est d'utiliser un service de type NodePort qui va utiliser un port de la machine hôte et retransmettra le traffic sur votre service.

On peut utiliser le port 30888 qui est ouvert sur l'ensemble d'INTERNET au niveau de la VM de TP.
On va délibérément fixer ce port mais on pourrait laisser Kubernetes en choisir un (il a une plage pour cela de 30000 à 32767)
fichier bgd.svc.nodeport.yml

```yaml
apiVersion: v1
kind: Service
metadata:
 name: bgd
spec:
 ports:
 - nodePort: 30888
   port: 8080
   protocol: TCP
   targetPort: 8080
 selector:
   app: bgd
 type: NodePort
```

Pour éviter les conflits, vous devriez supprimer le service bgd éventuellement existant ainsi que l'ingress associée.

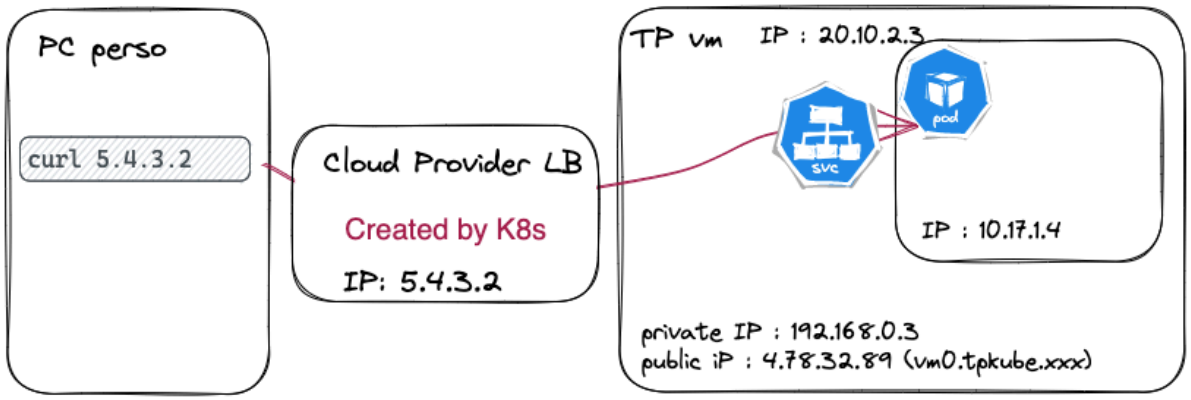
Vous pouvez tester d'appeler maintenant le service en utilisant le port exposé sur le host via NodePort :
`curl vmXX.tpcsonline.org:30888`

Vous remarquerez que cette solution n'est pas idéale surtout si plusieurs développeurs veulent réserver le même port sur le host, on va rapidement avoir des conflits.

Utilisation de services loadBalancer

BONUS

Si vous n'êtes pas en avance, vous pouvez sauter ce paragraphe.



Pour information, il existe une troisième manière (nous avons vu nodePort et Ingress) d'exposer un service à l'extérieur du cluster. Il s'agit du service type LoadBalancer.

Il s'agit d'une fonctionnalité particulière nécessitant une intégration CLOUD. On va donner à Kubernetes des éléments (souvent des credentials) pour qu'il puisse appeler l'API d'un cloud provider afin de provisionner un LoadBalancer chez le cloud provider et le configurer pour qu'il puisse accéder au service.

Dans notre cas, nous n'avons pas de cloud provider puisque nous hébergeons nous mêmes microk8s dans notre VM. Il existe pour information des projets (comme MetaLB) permettant de porter ce type de fonctionnalité dans un cloud privé mais cela dépasse largement le cadre de ce TP.

Retenez simplement que cela existe et que c'est assez pratique quand on utilise une distribution d'un cloud provider, il suffit de mettre type: LoadBalancer et notre service sera disponible depuis l'extérieur.
Toutefois, le revers de la médaille est que pour chaque service, vous aurez un LB différent et donc des coûts rapidement prohibitifs. (Rappel, l'ingress controller permet d'exposer des centaines/milliers de service depuis une seule instance)

Gérer des paramètres et des configurations depuis les containers

Comme pour les containers, on peut utiliser des variables d'environnements dans les pods pour disposer d'informations de configuration au moment du déploiement.

Nous allons explorer plusieurs manières de passer des paramètres à vos pods, pour cela nous allons poursuivre l'utilisation du descripteur de deployment simple que nous avons utilisé auparavant

Variables d'environnement

```
$ kubectl get pod -l app=bgd -o custom-columns=IP:.status.podIP
```

Nous allons changer la couleur du carré afficher par l'application via une variable d'environnement dont la valeur sera la couleur. L'image docker bgd est prévue pour afficher la valeur de couleur de la variable COLOR

éditer votre fichier de deployment et ajouter une directive env: dans la spec du container bgd (en dessous de name: , faites bien attention à l'indentation yaml)

```
  spec:
    containers:
    - image: quay.io/redhatworkshops/bgd:latest
      name: bgd
      env:
      - name: COLOR
        value: "green"
```

Appliquer votre descripteur et afficher votre application, vous devriez voir désormais une nouvelle couleur
Vous pouvez ensuite changer la valeur de la variable avec "blue" à la place de "green" et appliquer le descripteur.

Vous pouvez vous connecter dans le pod et réaliser un
`$ echo $COLOR`

On peut également passer dans des variables d'environnement des éléments provenant de la base etcd de Kubernetes, par exemple, vous pouvez ainsi disposer de l'adresse IP de votre POD dans une variable d'environnement (que vous pourriez utiliser directement depuis votre application)

```
  env:
  - name: POD_IP
    valueFrom:
        fieldRef:
          fieldPath: status.podIP
```

KQ3:

Que se passe-t-il exactement quand vous appliquez le descripteur de déploiement ? Est-ce que la couleur est mise à jour dans l'application quand vous appliquez le descripteur avec une nouvelle valeur pour la variable COLOR ? *(Attention, l'application ne supporte pas beaucoup de couleurs, celles qui sont sûres : green , yellow, blue)*

ConfigMap

Kubernetes possède bon nombre de ressources dont certaines liées au stockage d'informations. Nous allons utiliser les configMap qui permettent de stocker directement dans la base de données Kubernetes des informations

Nous allons utiliser cet exemple pour l'exercice (*fichier bgd.configmap.yml)*

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: bgd
data:
 COLOR: "yellow"
 index.html: |
   <html><body>
     <h1>This is a new page !</h1>
   <body></html>
```

Nous allons devoir mettre à jour notre descripteur de déploiement pour lui indiquer d'utiliser désormais la configMap pour la valeur de la variable COLOR

```
    env:
      - name: COLOR
        valueFrom:
          configMapKeyRef:
            name: bgd
            key: COLOR
```

BONUS : Aidez vous de la documentation (https://kubernetes.io/docs/concepts/configuration/configmap/) pour ajouter également le fichier que nous avons défini (index.html)

Voici une aide

```
   volumeMounts:
   - name: myvolume
     mountPath: "/usr/share/nginx/html"
     readOnly: true
 volumes:
 - name: myvolume
   configMap:
     name: bgd
```

KQ4:

Une fois la configMap avec la variable COLOR associée, est-ce que la couleur change dans l'application ?

Mettez à jour la valeur dans la configMap avec "blue", que se passe-t-il ? Que devez-vous faire en plus pour que la couleur change dans l'application ?

BONUS : Concernant le fichier monté comme un volume, quelles sont les autorisations sur les fichier qui sont montés (commande ls -l une fois exec dans le container)

BONUS : Pouvez-vous ajouter un fichier dans le répertoire où il y a index.html depuis le container ? Pouvez-vous modifier le fichier index.html ?

BONUS : Si vous utilisez une image nginx à la place de l'image bgd, que se passera-t-il en lançant la page d'accueil ?

Secrets

On peut utiliser les ressources secrets de la même manière que les configMaps. Leur intérêt étant que ces informations ne sont pas accessibles suivant les RBAC (autorisations) pour un utilisateur donné.

Attention, malgré le nom de secret, les données ne sont pas chiffrées mais simplement encodées. Toutefois, le fait de séparer les données sensibles dans une ressource dédiée est très pratique pour les manipuler et restreindre les droits sans toucher au reste de l'application.

https://kubernetes.io/docs/concepts/configuration/secret/

Nous allons commencer par utiliser le mode impératif de la CLI pour créer un secret et nous permettre fournir des données en clair (ie. non encodées en base64)

```
kubectl create secret generic mysecret \
      --from-literal=key1=supersecret --from-literal=key2=topsecret
```

Vous pouvez maintenant afficher le secret (ou même son contenu) avec les commandes describe et get

```
kubectl describe secret mysecret
kubectl get secret mysecret -oyaml
```

On peut bien sûr décrire un secret comme toute autre ressource et l'appliquer en utilisant la CLI et l'API Kubernetes.
On peut soit fournir des valeurs de clé dans le secret déjà encodées en base64 ou également fournir des données en clair en utilisant StringData.

Essayez de déployer l'exemple de secret ci-dessous (*fichier bgd.secret.yml*)

```yaml
apiVersion: v1
kind: Secret
metadata:
 name: bgd
type: Opaque
data:
 username1: YWRtaW4=
 password1: bXlTZWNyZXRQYXNzd29yZA==
stringData:
 config.yaml: |-
   apiUrl: "https://my.api.com/api/v1"
```

```
  username: {{username}}
  password: {{password}}
```

Inspirez-vous de l'exemple suivant pour monter le secret dans votre deployment

```
spec:
 containers:
 - name: mypod
   image: redis
   volumeMounts:
   - name: foo
     mountPath: "/etc/foo"
     readOnly: true
 volumes:
 - name: foo
   secret:
     secretName: mysecret
```

Dans l'exemple, vous remarquez une notation particulière pour certaines valeurs en stringData , par exemple `{{username}}`, ceci peut être utilisé dans une chaîne de CI/CD qui utilise du templating (par exemple jinja2) et viendra substituer les valeurs souhaitées (suivant l'environnement par exemple) au moment du déploiement.

Vous pouvez maintenant faire un exec bash dans le container et lister les éléments présents au niveau du point de montage que vous aurez choisi pour monter les secrets.
Utilisez ls et cat pour afficher le contenu des éléments.

KQ5:

Est-ce que les commandes describe et get vous permettent de voir les valeurs que vous avez entrées pour le sec ret mysecret ?

Si vous faites des get et describe sur le second secret créé via descripteur (bgd), est-ce que vous voyez cette fois des données en clair ? Que contient comme valeur les clés username1 et password1 (vous pouvez utiliser l'utilitaire base64 -d)

Listez les fichiers qui sont montés dans votre container en tant que secrets. Que remarquez-vous en termes de correspondance entre la description des secrets et leur implantation sous forme de fichiers ? Quel est le contenu des fichiers ?

Gestion des rollout de deployment

BONUS

Si vous n'êtes pas en avance, vous pouvez sauter ce paragraphe.

Un des atouts de Kubernetes est de s'occuper de toute la partie scalabilité et également la gestion des mises à jour d'une application.

Nous allons réaliser quelques exemples pour scaler notre application et la mettre à jour pour vérifier comment Kubernetes se comporte

Tout d'abord, nous allons déployer l'application suivante dans un deployment avec un service pour l'exposer :

fichier bgd.rollout.yml

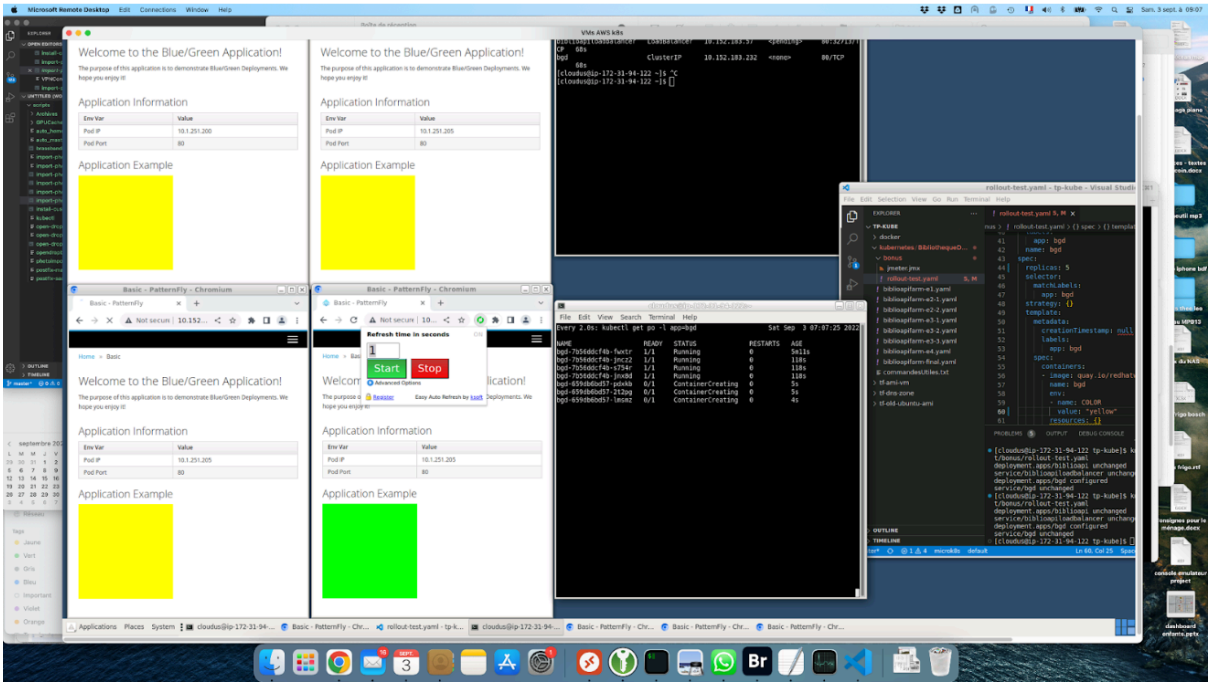
```
---
apiVersion: apps/v1
kind: Deployment
metadata:
labels:
app: bgd
name: bgd
spec:
replicas: 1
selector:
matchLabels:
app: bgd
strategy: {}
# strategy:
# type: Recreate
####
# strategy:
# type: RollingUpdate
####
# strategy:
# type: RollingUpdate
# rollingUpdate:
# maxSurge: 1
# maxUnavailable: 0
###
# strategy:
# type: RollingUpdate
# rollingUpdate:
# maxSurge: 0
# maxUnavailable: 20%
###
template:
metadata:
creationTimestamp: null
labels:
app: bgd
```

```
spec:
containers:
- image: quay.io/redhatworkshops/bgd:latest
name: bgd
env:
- name: COLOR
value: "green"
resources: {}
---
apiVersion: v1
kind: Service
metadata:
labels:
app: bgd
name: bgd
spec:
ports:
- port: 80
protocol: TCP
targetPort: 8080
selector:
app: bgd
---
```

Il s'agit d'indiquer à Kubernetes comment il doit se comporter quand il doit mettre à jour les pods qui composent le déploiement quand vous changez une partie de ses paramètres.

Ouvrez votre navigateur après déploiement sur l'IP du service kube (**`kubectl get service`** pour obtenir l'IP) correspondant à l'application bgd (c'est une application qui va afficher une couleur suivant ce qui est renseigné dans les paramètres)

Vous pouvez ouvrir plusieurs navigateurs et utiliser l'extension auto-refresh déjà préinstallée dans chromium avec un refresh de 1 secondes, ouvrez aussi une commande kubectl avec un watch sur get po (et le label de bgd).



Maintenant, vous pouvez jouer avec différents paramètres (en éditant votre déploiement) :

- Augmenter le nombre de replicas (défaut à 1 mais pour cet exercice passez le à 5 ou plus)
- “Version” de l’application : il s’agit d’éditer le déploiement pour changer la couleur :

```
        - name: COLOR
          value: "green"
```

- Vous pouvez également déployer volontairement une image qui ne fonctionne pas (pour voir ce qui se passe quand kubernetes déploie une version défectueuse suivant la stratégie de rollout)
    - Pour cela, mettez à jour l’image avec celle-ci : seb54000/tpkube-biblioapi:broken
- Enfin la stratégie de rollout (notamment vous pouvez tester recreate par exemple). Pour voir la différence suivant la stratégie utilisée, changer “la version” via la COLOR et surveiller l’évolution des pods (`watch kubectl get pods`)

Les différentes stratégies sont en commentaires dans le fichier de déploiement ci-dessus.

Il existe beaucoup d’articles sur les stratégies de déploiement, exemples :
https://spot.io/resources/kubernetes-autoscaling/5-kubernetes-deployment-strategies-roll-out-like-the-pros/
https://blog.container-solutions.com/kubernetes-deployment-strategies

La référence documentaire sur les déploiements :
https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

BONUS (KQ6) :
Comment le deployment peut-il gérer deux versions différentes de l'application sans se perdre ? *Regardez du côté des replicaSets durant une upgrade.*

Décrivez les différentes stratégies de rollout que vous avez pu utiliser et ce que vous observez quand vous changez de version (et notamment quand vous utilisez une version volontairement cassée)

Quels sont les avantages / inconvénients des différentes stratégies ? Et notamment, dans quel cas la stratégie recreate est adaptée pour une application ?

Enfin, imaginez un déploiement d'un micro service de 150 replicas sur un cluster dont la capacité maximum est de 200 pods, quelle solution utilisez-vous pour déployer une nouvelle version ? Deployment Rollout Strategy mais laquelle ? Autre solution à base de scripts maison ?

Gestion de la persistance (PVC)

https://kubernetes.io/fr/docs/concepts/storage/persistent-volumes/

Kubernetes propose la gestion de la persistance avec des configMap, des secrets comme nous l'avons vu mais il s'agit plutôt de données statiques que l'on ne peut pas facilement modifier par l'application elle-même.

Il y a donc également des ressources PersistentVolume qu'un administrateur peut créer manuellement en les hébergeant sur un backend de storage. Mais il existe mieux, un administrateur peut déclarer une StorageClass qui va dialoguer avec un driver CSI (Container Storage Interface) pour créer dynamiquement des PersistentVolumes lorsqu'un utilisateur du cluster Kubernetes en fera la demande via un PVC (PersistentVolumeClaim)

Le cluster microk8s dispose d'une storageClass déjà implémentée (et utilisant le storage local), nous allons donc créer un PVC et l'associer à notre deployment

```
kubectl get storageclass
```

Exemple de PVC dont vous pouvez vous inspirer pour en déployer un sur votre cluster (attention à ce que storageClassName corresponde à ce que vous avez sur votre cluster)
fichier bgd.pvc.yml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: bgd
spec:
 accessModes:
   - ReadWriteOnce
```

```
 resources:
   requests:
     storage: 100Mi
 storageClassName: microk8s-hostpath
```

Vous pouvez vérifier l'état de votre PVC et les actions qui ont été réalisées.
Dans notre exemple :
`kubectl get pvc bgd`
`kubectl get events`

Voici un exemple de montage d'un PVC dans la définition des containers au niveau d'un deployment, inspirez-vous de cet exemple pour ajouter le PVC que vous venez de créer sur votre deployment de test

```
spec:
 containers:
   - name: myfrontend
     image: nginx
     volumeMounts:
     - mountPath: "/var/www/html"
       name: mypd
 volumes:
   - name: mypd
     persistentVolumeClaim:
       claimName: myclaim
```

KQ7:

Juste après la création du PVC (et avant d'avoir réalisé le montage sur votre deployment), que voyez-vous dans les events ? Si vous regardez les détails de la storageClass pouvez-vous retrouver l'option qui explique ce comportement ?

Connectez vous dans le conteneur de votre déploiement et créez quelques fichiers dans le répertoire où vous avez monté votre volume. Supprimer ensuite les pods (kubectl delete pod) et vérifier que vos données sont toujours là après redémarrage.

Regarder les options de la storageClass et en particulier ReclaimPolicy qui sera utilisé en cas de suppression du PVC. Quels sont les avantages / inconvénients de la policy retain ?

Bilan intermédiaire : intérêt de Docker et Kubernetes

Prenons le temps de faire un rapide bilan de ce que nous avons vu et de se remémorer après avoir pratiqué quels sont les intérêts majeurs de ces technologies.

Souvenez-vous, il n'y a pas si longtemps, pour déployer une application écrite en Java accédant à une base de données et exposée via un ReverseProxy, il fallait :

- Développer et tester sur votre poste
- Rédiger un mode opératoire ou des scripts (incluant des informations sur les dépendances comme la version de Java nécessaire)
- Déployer sur un environnement d'intégration avec le risque de versions différentes
- Ajouter les composants comme le ReverseProxy pas forcément présent dans les environnements amonts

Via les containers, vous avez vu que l'on package dans un format unique l'application et ses dépendances. Concrètement, la version du runtime Java, ou des utilitaires systèmes nécessaires sont déjà dans le container via le DockerFile.

C'est cette même image de container que l'on pourra faire tourner sur n'importe quelle machine avec le même comportement.

Nous avons vu aussi que si l'on doit démarrer beaucoup de containers et s'assurer qu'ils fonctionnent toujours, les faire communiquer en réseaux, les rendre accessibles depuis l'extérieur ou y attacher des volumes pour gérer la persistance, on arrive vite à de nombreuses opérations avec Docker.

C'est donc Kubernetes qui va donner toute sa puissance aux containers en permettant de décrire via de simples fichiers yaml l'état désiré d'une architecture qui peut être très complexe, charge à l'orchestrateur de s'assurer du déploiement et ce tout au long du cycle de vie de l'application.

On peut reformuler cela ainsi, Kubernetes va permettre d'automatiser à l'échelle une bonne partie des actions réalisées sur Docker (notamment la mise en réseau) et la possibilité de décrire l'infrastructure de containers et sa configuration sous forme déclarative dans des fichiers de ressources yaml.
Il faut alors imaginer une architecture complexe avec des ReverseProxy pour la sécurité (intégrant pourquoi pas des fonctions WAF - Web Application Firewall), des frontaux pour le contenu statique, des backends pour les API, des bases de données, un cluster kafka pour le streaming de flux de données entre les services, …) avec toutes les difficultés qui vont avec :

- Difficulté pour installer
- Difficulté pour configurer
- Difficulté pour superviser
- Difficulté pour opérer et suivre dans le temps
- Difficulté pour scaler

Tout cela est simplifié par Kubernetes, mais ce n'est évidemment que le côté pile de la médaille. De l'autre côté c'est tout ce qui est facilité pour les développeurs

- Possibilité de reproduire la même infrastructure avec une plus petite empreinte pour tester
- Possibilité de créer des clones complets de toute l'architecture très simplement pour des tests ou autre besoins (capacité à reconstruire en cas d'incident majeur)
- Possibilité de totalement automatiser les actions de configuration et de déploiement
- Possibilité d'inclure des règles très fines au niveau de l'infra suivant les besoins et ce de façon totalement automatisée et sans passer par un mécanisme de ticket à une équipe gérant les serveurs et les OS
- Possibilité de modifier l'infrastructure directement par l'application (cette dernière pouvant bien sûr appeler elle-même l'API de Kubernetes)

Tout ce que nous n'avons pas encore vu sur Kubernetes

Avant de passer à la suite du TP, prenons quelques minutes pour lister des fonctionnalités ou concepts de Kubernetes pour lesquels nous n'avons malheureusement pas assez de temps à consacrer pour les expérimenter.

- Les namespaces, les RBAC, les quotas, les limit ranges
    - Ces éléments vont permettre de faire cohabiter une multitude de projets sur un même cluster en garantissant l'isolation des droits (RBAC), l'isolation des consommations de ressources (quotas et limit range)
- Les network Policies (dépendantes du SDN) - egress et ingress
    - Permettant de maîtriser les autorisations réseaux par namespace ou même par pod en utilisant simplement des labels (metadata) et surtout pas des adresses IP
- Les outils de supervision, logging, alerting
    - On peut citer particulièrement Prometheus, alertManager et Grafana et bien sûr la stack EFK (Elastic, FluentD, Kibana) mais aussi Loki
- Les outils de templating et déploiement (helm, kustomize, jsonet) et les pipelines de CI/CD (jenkins, tekton)
- Les opérateurs qui permettent d'étendre l'API de Kubernetes pour manager des ressources custom en mode déclaratif et en profitant des mécanismes de réconciliation
    - Les opérateurs permettent d'automatiser entièrement les actions d'installation, configuration et exploitation (day2) d'un système. Cela peut être très pratique et puissant pour des systèmes complexes comme des bases de données en cluster (https://cloudnative-pg.io/, https://github.com/mongodb/mongodb-kubernetes-operator), des clusters kafka (https://strimzi.io/) ou même des solutions datas (https://docs.stackable.tech/home/stable/hdfs/index.html)
- Les notion de service mesh pour manager une architecture micro-service (sécurité, scaling de l'exposition, gestion des upgrades et des versions en parallèle)
    - Une des distribution majeures de service Mesh est bien sûr ISTIO qui propose des tutorials : https://istio.io/latest/docs/setup/getting-started/ (avec sampleApp)

Pour continuer l'entraînement sans forcément installer un kube, vous pouvez utiliser des services tels que KillerCoda :

https://killercoda.com/playgrounds/scenario/kubernetes

Si vous souhaitez également installer et retravailler des éléments docker ou kubernetes sur votre poste, vous pouvez parti de docker desktop qui est très simple à utiliser et installer dans un environnement windows

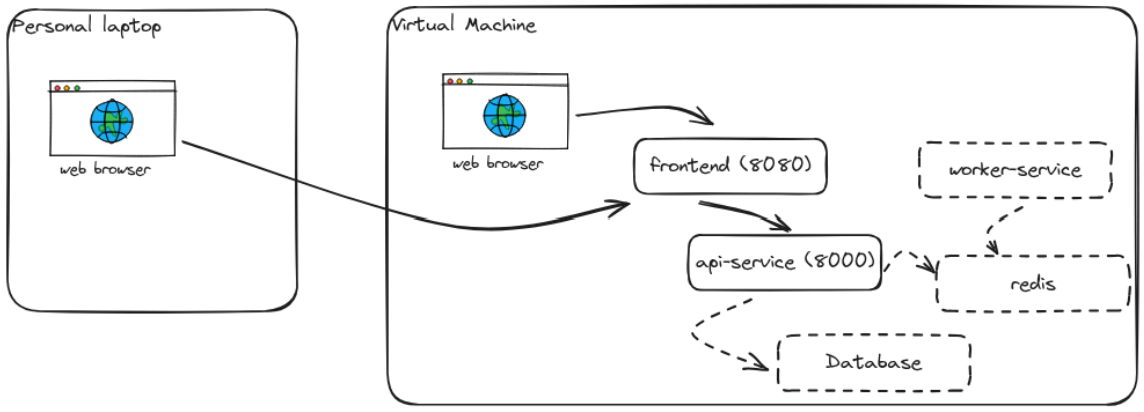
https://docs.docker.com/desktop/kubernetes/

Déployer l'application demoboard sur Kubernetes

Nous allons illustrer tout ce que nous avons vu en déployant une application de démonstration qui permet d'ajouter (et supprimer) des tâches fictives dans une liste et éventuellement de lancer un traitement associé à ces tâches en mode asynchrone et afficher le statut de ces traitements.

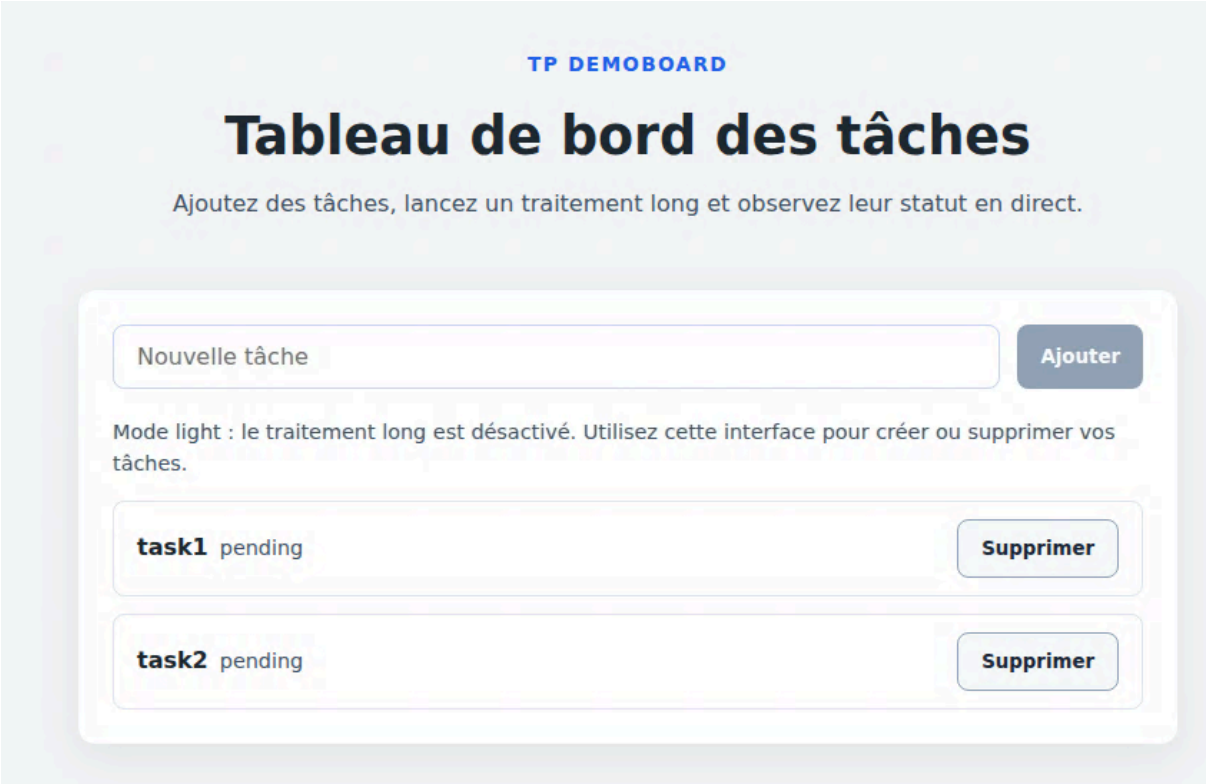
Il s'agit de la même application que celle utilisée dans le TP docker
Voici quelques rappels et illustrations sur l'application.

Architecture globale de l'application



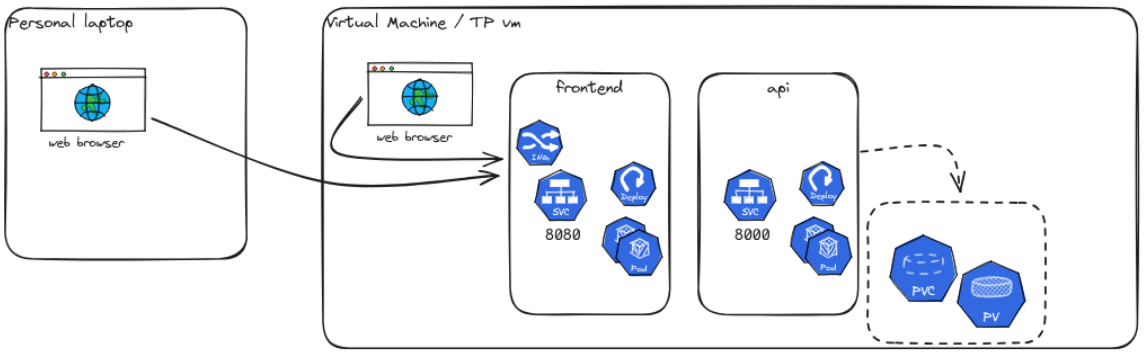
Exemple de screenshots de l'application

Mode simple (sans la partie traitement)



Mode complet (avec possibilité de lancer des traitements asynchrones)

Architecture Kubernetes (mode simple)



Builder et pusher les images pour Kubernetes (mode simple)

Dans le TP docker, vous avez réalisé des tests, builder des images en faisant des modifications pour un fonctionnement docker local.
Nous avons maintenant besoin de vérifier et rebuilder les images pour l'environnement Kubernetes. Il faut également pusher nos images sur notre registry local afin que Kubernetes puisse les récupérer.

Pour tout ce qui concerne la vérification, build et push des images nous allons reprendre les travaux dans le répertoire précédemment utilisé :
`docker/demoboard/demoboard-source/`

Souvenez-vous du fichier `frontend-service/docker/nginx.conf` , si vous avez réalisé l'ensemble du TP docker, vous l'avez normalement modifier pour pointer la directive proxy_pass en cohérence avec le déploiement docker local.

Dans le cadre d'un déploiement kubernetes, vous allez bénéficier des ingress et surtout des services kubernetes. Ainsi, vous allez pouvoir déployer conformément au schéma kubernetes ci-dessus et exposer votre api-service avec un service kubernetes qui je vous le rappelle dispose automatiquement d'un record DNS interne au sein du namespace et du cluster kubernetes.

Vous pouvez donc revenir à la configuration standard dans ce fichier pour le proxy_pass avec : `proxy_pass http://api-service:8000/;`

Nous allons pour le moment builder uniquement les images du frontend et de l'api (déploiement du mode simple). Lancer le build avec un tag intégrant notre registry externe puis pusher les images sur le registry.

Le frontend prend un argument pour forcer le mode simple :
VITE_ENABLE_WORKER="false"
Les commandes suivantes fonctionnent depuis le répertoire
`docker/demoboard/demoboard-source/`

```
docker build --build-arg VITE_ENABLE_WORKER="false" -t
localhost:32000/front:v1 ./frontend-service
docker build -t localhost:32000/api:v1 ./api-service
docker push localhost:32000/front:v1
docker push localhost:32000/api:v1
```

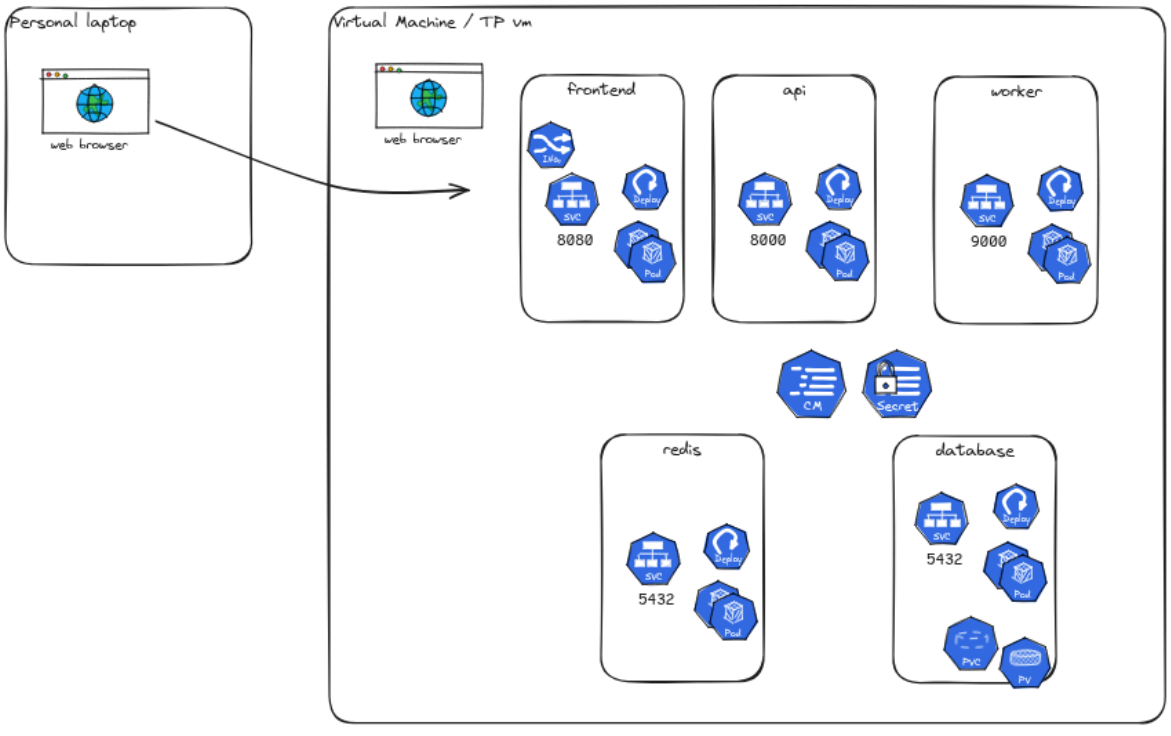
Déployer demoboard sur Kubernetes (mode simple)

Aller dans le sous répertoire `kubernetes/demoboard`

Regarder le contenu du fichier demoboard-light.yaml, regarder si vous devez adapter la règle ingress (notamment pour le host) et vous pouvez ensuite l'appliquer sur votre cluster `kubectl apply -f demoboard-light.yaml`

KQ8 :
Est-ce que les pods fonctionnent bien ? Quelles sont les adresses IP des services ? Essayer d'atteindre l'application depuis le navigateur de la VM avec l'IP du service, le record DNS associé à l'ingress (et essayer aussi depuis votre poste local)

Architecture Kubernetes (mode complet)



Builder et pusher les images pour Kubernetes (mode complet)

Aller dans le sous répertoire **`docker/demoboard`**

On doit en plus du worker builder aussi l'image front pour le mode complet, on fera un nouveau tag v2 pour celui-ci (en ne précisant plus d'argument VITE_ENABLE_WORKER ou sinon en le mettant à true)

```
docker build -t localhost:32000/front:v2 ./frontend-service
docker push localhost:32000/front:v2
docker build -t localhost:32000/worker:v1 ./worker-service
docker push localhost:32000/worker:v1
```

Déployer demoboard sur Kubernetes (mode complet)

Aller dans le sous répertoire **`kubernetes/demoboard`**

Regarder le contenu du fichier demoboard-full.yaml, regarder si vous devez adapter la règle ingress et vous pouvez ensuite l'appliquer sur votre cluster. Vérifier bien que les différentes images utilisent les bons tags si vous n'avez pas strictement suivi les tags proposés

```
kubectl apply -f demoboard-full.yaml
```

Attention, si vous avez toujours l'ingress pour demoboard-light déployée avec le même host, il y a un risque de conflit et que ce ne soit pas pris en compte, vous devriez supprimer d'abord l'ingress de demoboard-light (**`kubectl delete ingress -n demoboard-light demoboard-light`**)

Connectez-vous à nouveau à l'application depuis la VM de travail ou depuis votre PC personnel, est-ce que vous pouvez maintenant lancer les traitements ? Vous avez bien l'application en mode complet ?

KQ9 :
Vous pouvez bien sûr tester de scaler les replicas pour ajouter des instances du frontend, de l'api ou du worker et vérifier si le comportement de l'application est correct.

Vous pouvez regarder les logs des différents pods pour vérifier lesquels reçoivent effectivement les requêtes si vous avez augmenter le nombre de replicas.

KQ10 :
Regarder en détail le fichier demoboard-full.yaml et essayer de l'améliorer pour factoriser les informations de connexion à la base de données et à REDIS.

Vous pouvez mettre les passwords dans un secret partagé et les informations type username, port ou host dans une configmap
Inspirez vous de la documentation Kubernetes pour définir des variables d'environnement

dans le déploiement directement depuis les valeurs de la configMap ou du secret (valueFrom)

Utilisation des clusters eks (multi-nodes)

Un ou plusieurs (suivant le nombre d'étudiants)) clusters eks (AWS Elastic Kubernetes Service) sont déployés et accessibles pour vous permettre d'autres tests et aller plus loin.

Chacun des clusters eks a la configuration suivante :

- Une URL d'accès spécifique (générée par AWS)
- Un compte cluster-admin avec fichier kube-config associé
- Un compte des service par étudiant (vmXX) avec fichier kube-config associé
- Un namespace par étudiant (vmXX) avec le role admin associé au compte de service
- Un ingress controller nginx (exposé via LB) et un record wildcard par étudiant et par cluster eks pour exposer vos ingress rules (*.vmXX-svc.eksYY..tpcsonline.org)

Tout est déjà pré-configuré sur vos vms pour vous permettre d'utiliser le plus facilement possible ces clusters eks et en particulier :

- un alias linux 'eks' qui va utiliser le fichier kube-config avec le compte de service et le namespace étudiant (vmXX)
    - Vous pouvez lister les différents clusters eks disponibles
    - `eks config get-contexts`
    - et en sélectionner un différent de celui par défaut (qui est toujours le premier)
    - `admin-eks config use-context cluster01-admin`

Il faut en revanche que vos images custom (celles de demoboard) soient bien accessibles depuis le cluster eks. Malheureusement l'utilisation du container registry local serait trop complexe (eks ne fait du pull que en HTTPS obligeant à déployer des certificats publiques sur registry local).
A la place, nous allons utiliser le registry AWS ECR (Elastic Container Registry) mais ne vous inquiétez pas tout est préconfiguré pour vous.

Vous pouvez simplement taguer vos images existantes et les push vers le registry ecr. Il faut juste dans le tag bien utiliser le repo name ecr (et URL) spécifique qui est créé pour chaque étudiant.

Un fichier est disponible sur la VM pour vous donner l'URL de votre repo ecr ainsi que des exemples de commandes :
`${HOME}/.kube/ecr_access_info.txt`
L'URL du repo ecr est également normalement disponible dans le tableau récap des vms étudiantes : https://docs.tpcsonline.org/vms.html

Attention, sur ecr, vous n'avez qu'un repository par étudiant au sein du registry. Pour rappel les images sont push avec le nom du repository, vous ne pourrez donc pas les "nommer" front:v1 , api:v2 mais <repo_name>:front-v1 par exemple.

```
docker image tag localhost:32000/front:v1 <url_and_repo>:front-v1
docker push <url_and_repo_name>:front-v1
```

Le repo ecr nécessite une authentification mais pour simplifier une configuration a été fait pour vous rendant cela automatique (pour les curieux, c'est basé sur amazon-ecr-credential-helper et nous avons un script custom pour charger dans le cas du helper un profil aws spécifique avec les credentials pour récupérer un token ecr : /usr/local/bin/docker-credential-ecr-login-ecr)

Un dernier point, des RBAC ont été ajoutés pour qu'en tant que admin de votre namespace, vous puissiez lister certaines ressources cluster-wide comme les nodes (et leur labels), les pv (persistentVolumes) et les sc (storageClasses), vous pouvez tester :

```
eks get nodes --show-labels
eks get pv
eks get sc
```

Vous pouvez donc dès maintenant utiliser un ou plusieurs clusters eks pour aller plus loin et en particulier profiter de la présence de plusieurs nodes.

Parmi les exercices que vous pouvez réaliser :

- Déployer l'application demoboard en adaptant les fichiers de ressources (notamment pour la création des pvc , vérifier le nom de la storage class employée, vous devrez aussi adapter l'ingress host en utilisant le wildcard DNS qui est mis en place par cluster eks : *.vmXX-svc.eksYY..tpcsonline.org)
- Déployer l'application demoboard avec 3 replicas pour le front et l'api en vous assurant que chacun des replicas soit toujours sur une AZ différente (chaque node est sur une AZ différente)
    - Vous devez vous appuyer sur les labels standards des nodes (toplogy) et consulter la documentation des topology constraints : https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/
- Vous pouvez déployer un opérateur pour la gestion des bases de données mySQL ou PostgreSQL par exemple : https://operatorhub.io/?category=Database
    - Tester ensuite le déploiement d'un cluster avec des noeuds sur chaque AZ et des scénarios de maintenance ou failover automatique

Prendre un peu de recul

Nous avons vu beaucoup de concepts et réalisé des manipulations permettant de visualiser le comportement et l'usage de ces technologies.

Vous avez ainsi mieux compris ce que permettent les containers et un orchestrateur.

KQ11:

Réfléchissez à une expérience récente (ou non) personnelle ou professionnelle où vous pensez que vous pourriez (ou auriez dû) utiliser Docker ou Kubernetes pour être plus efficace / rapide / agile / …

Décrivez en quelques lignes ce projet et en quoi l'utilisation de containers (avec orchestration ou non) est bénéfique dans votre cas.