Cahier de TP
« Maillage de service avec Istio »

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux
- Éditeur .yaml : VSCode ou autre
- Docker, Git
- Kubernetes, Helm
- JMeter

Attention, il est recommandé d'installer ***kind*** *ou* ***docker desktop*** sur Windows: distributions kubernetes plus légères que *minikube*

Installation *kind* :

https://kind.sigs.k8s.io/docs/user/quick-start/#installation

Table des matières

1: Démarrer avec Istio

1.1 Installation profil demo + addons

Téléchargement distribution

- Installation Istio avec une commande Shell :

```
curl -L https://istio.io/downloadIstio | sh -
```

- Sinon, télécharger manuellement la distribution à

https://github.com/istio/istio/releases

Ajouter le client au PATH

```
export PATH=$PWD/bin:$PATH
```

Installer Istio dans kubernetes

Après avoir vérifié que votre cluster s'exécute :

```
istioctl install --set profile=demo -y
```

Vérifier ensuite l'installation avec

```
istioctl verify-install
```

Installer les composants du control plane

```
kubectl apply -f ./samples/addons
```

Vérifier les pods qui s'exécutent :

```
$ kubectl get pod -n istio-system
grafana
istio-egressgateway
istio-ingressgateway
istiod
jaeger
kiali
prometheus (2/2)
```

1.2 Déployer une application avec « Istio enabled »

Créer un namespace ***formation*** et l'activer comme namespace par défaut

```
kubectl create namespace formation
```

```
kubectl config set-context $(kubectl config current-context) \
--namespace=formation
```

Pour visualiser les effets d'Istio sur une ressources applicatives Kubernetes, exécuter :

```
istioctl kube-inject -f TP_Data/1_Demarrer/1.2/catalog.yaml
```

Observer les ajouts dans les ressources kubernetes :

Activer ensuite l'injection automatique d'istio pour le namespace *formation*

```
kubectl label namespace formation istio-injection=enabled
```

Créer le déploiement de l'application ***catalog***

```
kubectl apply -f TP_Data/1_Demarrer/1.2/catalog.yaml
```

Vérifier

```
$ kubectl get pod
NAME                       READY     STATUS    RESTARTS  AGE
catalog-7c96f7cc66-flm8g 2/2       Running   0         1m
```

On peut vérifier l'accès au service à partir du cluster Kubernetes via :

```
kubectl run -i -n default --rm --restart=Never dummy \
--image=curlimages/curl --command -- \
sh -c 'curl -s http://catalog.formation/items/1'
```

Déployer ensuite le service ***webapp*** :

```
kubectl apply -f TP_Data/1_Demarrer/1.2/webapp.yaml
```

Exposer le service sur un port local pour y accèder

```
kubectl port-forward deploy/webapp 8080:8080
```

1.3 Parcours des fonctionnalités

Utiliser la gateway de Istio pour exposer le service

```
kubectl apply -f TP_Data/1_Demarrer/1.2/ingress-gateway.yaml
```

Accéder à des micro-services via la gateway :
minikube

```
export INGRESS_PORT=$(kubectl -n istio-system get service istio-
ingressgateway -o jsonpath='{.spec.ports[?
```

```
(@.name=="http2")].nodePort}')
export SECURE_INGRESS_PORT=$(kubectl -n istio-system get service
istio-ingressgateway -o jsonpath='{.spec.ports[?
(@.name=="https")].nodePort}')
export INGRESS_HOST=$(minikube ip)
```

Dans un autre terminal :

```
minikube tunnel
```

kind

```
export INGRESS_HOST=$(kubectl get po -l istio=ingressgateway -n
istio-system -o jsonpath='{.items[0].status.hostIP}')
export INGRESS_PORT=$(kubectl -n istio-system get service istio-
ingressgateway -o jsonpath='{.spec.ports[?
(@.name=="http2")].nodePort}')
export SECURE_INGRESS_PORT=$(kubectl -n istio-system get service
istio-ingressgateway -o jsonpath='{.spec.ports[?
(@.name=="https")].nodePort}')
```

Puis dans un navigateur :

```
http://$INGRESS_HOST:$INGRESS_PORT/api/catalog
```

Docker Desktop

Le défaut est :
`http://localhost:80`

Enfin si on ne peut pas utiliser un LoadBalancer

```
kubectl port-forward deploy/istio-ingressgateway \
-n istio-system 8080:8080
```

Observabilité

Visualiser les tableaux de bord Grafana

Utiliser *istioctl* pour faire un port-forward vers Grafana

```
$ istioctl dashboard grafana
```

Simuler de la charge avec

```
while true; do curl http://$INGRESS_HOST:
$INGRESS_PORT/api/catalog; sleep .5; done
```

Visualiser les requêtes avec *Jaeger*

```
istioctl dashboard jaeger
```

Résilience

Générons 100 % d'erreur sur le service catalog

```
TP_Data/1_Demarrer/1.3/chaos.sh 50 100
```

Vérifier avec

```
curl -v http://$INGRESS_HOST:$INGRESS_PORT/api/catalog
```

Générer maintenant 50 % d'erreur

```
TP_Data/1_Demarrer/1.3/chaos.sh 50 50
```

Vérifer

```
while true; do curl http://$INGRESS_HOST:$INGRESS_PORT/api/catalog
; \
sleep .5; done
```

Appliquer une politique de retry

```
kubectl apply -f TP_Data/1_Demarrer/1.3/catalog-
virtualservice.yaml
```

Réexécuter et on devrait voir moins d'erreur

Supprimer le chaos

```
TP_Data/1_Demarrer/1.3/chaos.sh 500 delete
```

Contrôle du trafic

Commencer par déployer une deuxième version du service *catalog*

```
kubectl apply -f TP_Data/1_Demarrer/1.3/catalog-deployment-v2.yaml
```

Pour distinguer les 2 versions, déployer une ***DestinationRule*** Istio

```
kubectl apply -f TP_Data/1_Demarrer/1.3/catalog-
destinationrule.yaml
```

Dans un premier temps, on route toutes les requêtes vers la v1

```
kubectl apply -f TP_Data/1_Demarrer/1.3/catalog-virtualservice-
all-v1.yaml
```

Tester avec :

```
while true; do curl http://$INGRESS_HOST:INGRESS_PORT/api/catalog;
sleep .5; done
```

Puis mettre à jour le VirtualService catalog en définissant une nouvelle route :

```
kubectl apply -f TP_Data/1_Demarrer/1.3/catalog-virtualservice-
dark-v2.yaml
```

Tester avec

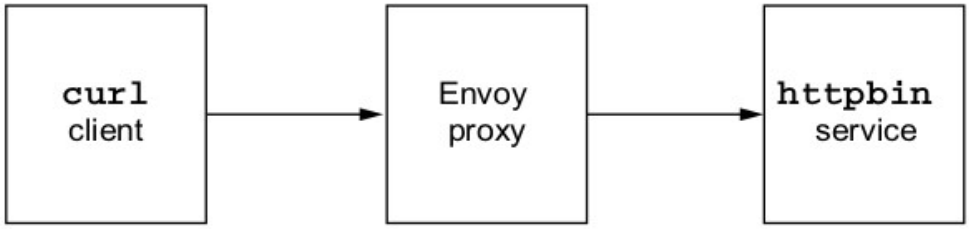
```
curl http://$INGRESS_HOST:INGRESS_PORT/api/catalog
```

Puis :

```
curl http://$INGRESS_HOST:INGRESS_PORT/api/catalog \
   -H "x-dark-launch: v2"
```

La réponse est sensiblement différente.

1.4 Envoy



Récupérer les 3 images suivantes :

```
docker pull envoyproxy/envoy:v1.19.0
docker pull curlimages/curl
docker pull citizenstig/httpbin
```

Démarrer httbin qui est un service qui renvoie les détails sur la requête qu'il a reçu

```
docker run -d --name httpbin citizenstig/httpbin
```

Effectuer une requête avec

```
docker run -it --rm --link httpbin curlimages/curl \
curl -X GET http://httpbin:8000/headers
```

Démarrer Envoy en lui passant la configuration :

```
docker run --name proxy --link httpbin envoyproxy/envoy:v1.19.0 \
--config-yaml "$(cat TP_Data/1_Demarrer/1.4/simple.yaml)"
```

Faire une requête vers le proxy :

```
docker run -it --rm --link proxy curlimages/curl \
curl -X GET http://proxy:15001/headers
```

Envoy a généré 2 nouvelles entêtes :

- ***X-Request-Id*** , utilisé pour corréler les requêtes qui nécessitent plusieurs hops (tracing)
- ***X-Envoy-Expected-Rq-Timeout-Ms*** qui peut être utilisé pour interrompre une requête trop longue

Appel de l'API admin

```
docker run -it --rm --link proxy curlimages/curl \
curl -X GET http://proxy:15000/
```

Liste les endpoints disponibles :

- */certs* Certificats de la machine
- */clusters* : Clusters configurés
- */config_dump* : La configuration de Envoy
- */listeners* : Les Listeners configurés
- */logging* : La conf des traces
- */stats* : Les stats
- */stats/prometheus* : Les stats au format Prometheus

```
docker run -it --rm --link proxy curlimages/curl \
curl -X GET http://proxy:15000/stats
```

Utiliser maintenant une autre configuration qui ajoute une politique de ré-essai par rapport à la précédente configuration

```
docker rm -f proxy
docker run --name proxy --link httpbin envoyproxy/envoy:v1.19.0 \
--config-yaml "$(cat TP_Data/1_Demarrer/1.4/simple_retry.yaml)"
```

Accéder à une URL provoquant une erreur 500

```
docker run -it --rm --link proxy curlimages/curl \
curl -v http://proxy:15001/status/500
```

Visualiser les stats concernant les ré-essais ;

```
docker run -it --rm --link proxy curlimages/curl \
curl -X GET http://proxy:15000/stats | grep retry
```

2. Istion in Action

2.1 Gateway

Nettoyer les ressources Istio du 1er atelier

```
kubectl delete deployment,svc,gateway,\
virtualservice,destinationrule --all -n formation
```

Créer une ressource *Gateway*

```
kubectl -n formation apply -f \
TP_Data/2_IstioInAction/2.1_Gateway/coolstore-gw.yaml
```

Vérifier la configuration du proxy avec :

```
istioctl proxy-config route deploy/istio-ingressgateway \
-o json --name http.8080 -n istio-system
```

Créer une ressource *VirtualService*

```
kubectl apply -n formation -f
TP_Data/2_IstioInAction/2.1_Gateway/coolstore-vs.yaml
```

Si nécessaire redémarrer les services

```
kubectl config set-context $(kubectl config current-context) \
--namespace=istioinaction
kubectl apply -f ./TP_Data/1_Demarrer/1.2/catalog.yaml
kubectl apply -f ./TP_Data/1_Demarrer/1.2/webapp.yaml
```

Essayer d'accéder avec une commande *curl*, il est nécessaire de préciser l'entête ***Host*** pour que le routage correct s'effectue.

```
curl http://$INGRESS_HOST:$INGRESS_PORT/api/catalog -H "Host:
webapp.formation.io"
```

Mise en place de TLS

Création du secret ***webapp-credential***

```
kubectl create -n istio-system secret tls webapp-credential \
--key TP_Data/2_IstioInAction/2.1_Gateway/webapp.formation.io.key \
--cert TP_Data/2_IstioInAction/2.1_Gateway/webapp.formation.io.crt
```

Application de TLS sur la gateway

```
kubectl apply -n formation -f TP_Data/2_IstioInAction/2.1_Gateway/coolstore-gw-
```

```
tls.yaml
```

Pour accéder en https, modifier votre fichier ***hosts*** afin que ***webapp.formation.io*** soit résolu en *$INGRESS_HOST*

Utiliser firefox pour accéder au port *https* et accepter le certificat « *self-signed* »

Redirection *http* vers *https*

```
kubectl apply -f TP_Data/2_IstioInAction/2.1_Gateway/coolstore-gw-tls-redirect.yaml
```

On peut voir la redirection avec :

```
curl -v http://172.23.0.2:31087/api/catalog -H "Host: webapp.formation.io"
```

2.2 Routing

2.2.1 Routing à partir de la gateway

Nettoyer les ressources Istio du 1er atelier

```
kubectl delete deployment,svc,gateway,\
virtualservice,destinationrule --all -n formation
```

Déploiement de la v1

```
kubectl apply -f ./TP_Data/1_Demarrer/1.2/catalog.yaml
```

Exposer via une gateway et router via un Virtual Service

```
kubectl apply -f ./TP_Data/2_IstioInAction/2.2_Routing/catalog-gateway.yaml
kubectl apply -f ./TP_Data/2_IstioInAction/2.2_Routing/catalog-vs.yaml
```

Vérifier que le service accessible avec :

```
curl http://$INGRESS_HOST:$INGRESS_PORT/items -H "Host: catalog.formation.io"
```

Définition des subsets

Créer une *DestinationRule* qui définit les sous-ensemble v1 et v2

```
kubectl apply -f ./TP_Data/2_IstioInAction/2.2_Routing/catalog-dest-rule.yaml
```

Mettre à jour le *VirtualService* afin qu'il route le trafic vers le sous-ensemble v1

```
kubectl apply -f ./TP_Data/2_IstioInAction/2.2_Routing/catalog-vs-v1.yaml
```

Vérifier que la v1 est accessible avec :

```
curl http://$INGRESS_HOST:$INGRESS_PORT/items -H "Host: catalog.formation.io"
```

Déploiement de la v2

```
kubectl apply -f ./TP_Data/1_Demarrer/1.3/catalog-deployment-v2.yaml
```

Vérifier que seule la v1 est accessible avec :

```
for in in {1..10}; do curl http://$INGRESS_HOST:$INGRESS_PORT/items \
-H "Host: catalog.formation.io"; printf "\n\n"; done
```

Définition du routing des clients ayant l'entête ***x-istio-cohort:Internal*** vers la v2

Visualiser puis appliquer ***catalog-vs-v2-request.yaml***

Tester l'accès à la v2 avec :

```
curl http://$INGRESS_HOST:$INGRESS_PORT/items \
-H "Host: catalog.formation.io" -H "x-istio-cohort: internal"
```

2.2.2 Routing interne

Nettoyer les ressources Istio:

```
kubectl delete gateway,virtualservice,destinationrule –all -n formation
```

Recréer l'architecture avec :

- Le service *webapp* et les 2 versions de *catalog*
- 1 *gateway* exposant le port 80 à l'extérieur et routant vers un *virtual service*
- 1 *virtual service* permettant d'atteindre *webapp*

Créer un *VirtualService* et une *DestinationRule* qui route tous le trafic vers la v1 de *catalog*.

Dans un 2ème temps, : Faire en sorte que 10 % des requêtes soient routées vers la v2

2.2.3 Canary déploiement avec *Flaeger*

```
kubectl delete deploy catalog-v2
kubectl delete service catalog
kubectl delete destinationrule catalog
```

Installer *prometheus*

```
kubectl apply -f $ISTIO_HOME/samples/addons/prometheus.yaml \
-n istio-system
```

Installer Flaeger

```
helm repo add flagger https:/./flagger.app
kubectl apply -f \
https://raw.githubusercontent.com/fluxcd/\
flagger/main/artifacts/flagger/crd.yaml
helm install flagger flagger/flagger \
--namespace=istio-system \
--set crd.create=false \
--set meshProvider=istio \
--set metricsServer=http://prometheus:9090
```

Vérifier les déploiements de *Prometheus* et *Flaeger* dans l'espace de nom ***istio-system***

```
kubectl get po -n istio-system
```

Appliquer la configuration du canary pour le service ***catalog***

```
kubectl apply -f ./TP_Data/2_IstioInAction/2.2_Routing/flagger/catalog-release.yaml
```

Surveiller le déploiement avec

```
kubectl get canary catalog-release -w
```

Flagger a automatiquement créé les ressources Kubernetes nécessaires : *Deployment* , *Service* et *VirtualService*

Vérifier la configuration du *VirtualService* :

```
kubectl get virtualservice catalog -o yaml
```

100% du trafic doit être routé vers le service ***catalog-primary*** et 0% vers le canary

Flagger surveille les modifications apportées au déploiement d'origine (dans ce cas, catalog), crée le déploiement Canary (*catalog-canary*) et le service (*catalog-canary*) et ajuste les poids du VirtualService

Déploiement de la v2

Pendant le déploiement, générer des requêtes avec

```
while true; do curl "http://$INGRESS_HOST:$INGRESS_PORT/api/catalog" \
-H "Host: webapp.formation.io" ; sleep 1; done
```

Appliquer le déploiement v2

```
kubectl apply -f TP_Data/2_IstioInAction/2.2_Routing/flagger/catalog-deployment-v2.yaml
```

Et surveiller avec :

```
kubectl get canary catalog-release -w
```

Après avoir atteint 100 % vers la v2. Effectuer un nettoyage :

```
kubectl delete canary catalog-release
kubectl delete deploy catalog
kubectl apply -f services/catalog/kubernetes/catalog-svc.yaml
kubectl apply -f services/catalog/kubernetes/catalog-deployment.yaml
kubectl apply -f services/catalog/kubernetes/catalog-deployment-v2.yaml
kubectl apply -f ch5/catalog-dest-rule.yaml

helm uninstall flagger -n istio-system
```

2.2.4 Service externe

Bloquer le trafic externe :

```
istioctl install --set profile=demo \
--set meshConfig.outboundTrafficPolicy.mode=REGISTRY_ONLY
```

Installer l'application ***forum***

```
kubectl apply -f
TP_Data/2_IstioInAction/2.2_Routing/serviceentry/forum-all.yaml
```

Essayer d'atteindre le service via :

```
curl http://$INGRESS_HOST:$INGRESS_PORT/api/users -H "Host: webapp.formation.io"
```

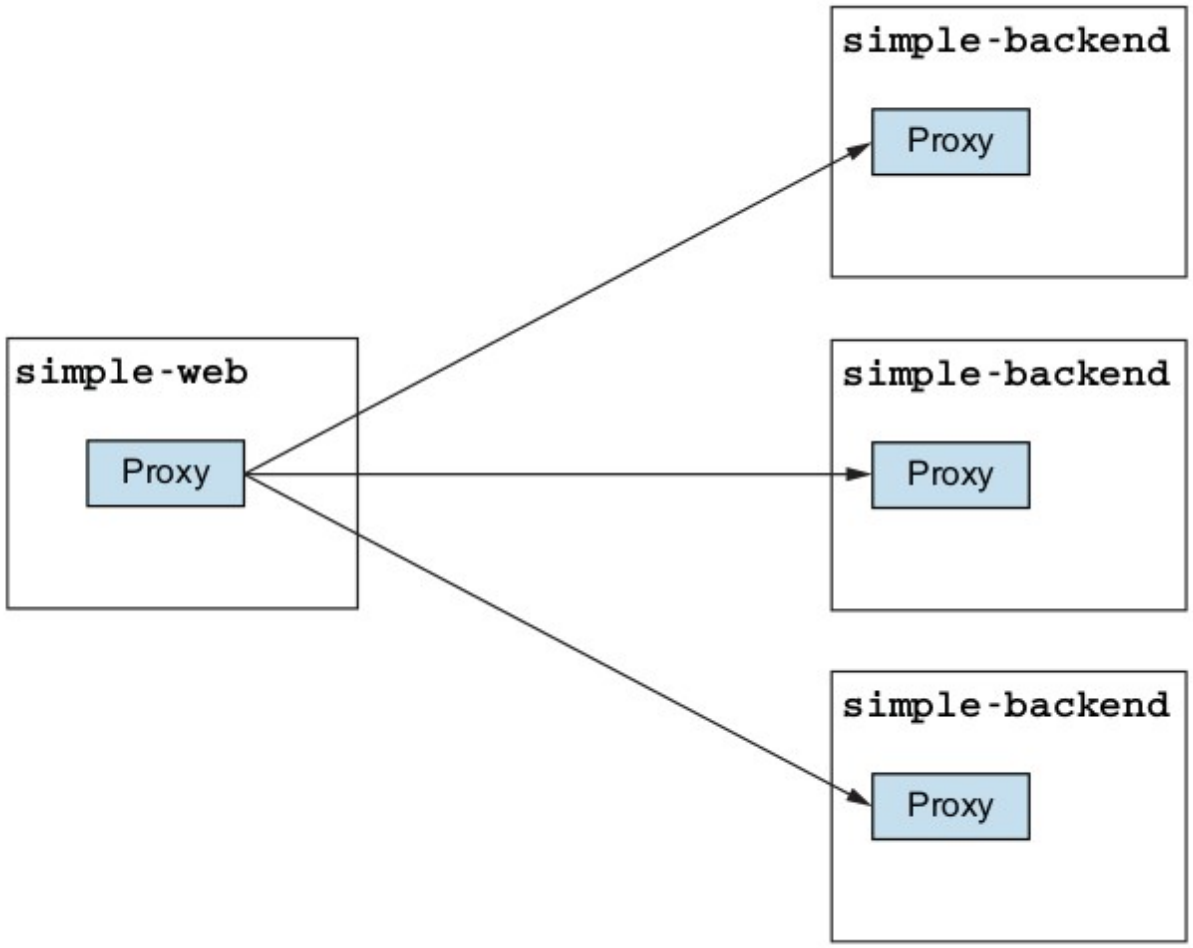
Ensuite déployer la ressource ***ServiceEntry***

```
kubectl apply -f TP_Data/2_IstioInAction/2.2_Routing/serviceentry/forum-
serviceentry.yaml
```

Re-tenter la commande *curl*

2.3 Résilience

Architecture utilisée pour cet atelier :



2.3.1 Répartition de charge

Faire le nettoyage

```
kubectl config set-context $(kubectl config current-context) \
--namespace=formation
kubectl delete virtualservice,deployment,service,\
destinationrule,gateway –all
```

```
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.1_ClientLoadBalancing/simple-
backend.yaml
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.1_ClientLoadBalancing/simple-web.yaml
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.1_ClientLoadBalancing/simple-web-
gateway.yaml
```

Déployer une ressource *DestinationRule* qui applique une répartition **ROUND_ROBIN**

```
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.1_ClientLoadBalancing/simple-backend-
dr-rr.yaml
```

```
curl -s -H "Host: simple-web.formation.io" http://$INGRESS_HOST:$INGRESS_PORT/
```

La réponse affiche la chaîne d'appels

Visualisez alors la répartition avec :

```
for in in {1..10}; do \
curl -s -H "Host: simple-web.formation.io" $INGRESS_HOST:$INGRESS_PORT \
| jq ".upstream_calls[0].body"; printf "\n"; done
```

On déploie ensuite une version de ***simple-backend-1*** qui ajoute de la latence (jusqu'à 1 seconde)

Utiliser le script JMeter fourni pour générer de la charge. Noter les résultats obtenus

Modifier ensuite la *DestinationRule* pour appliquer l'algorithme ***LEAST_CONN***

Refaire un tir de charge avec JMeter et comparer les résultats obtenus

2.3.2 Timeout et Réessai

Réinitialiser l'environnement :

```
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.1_ClientLoadBalancing/simple-web.yaml
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.1_ClientLoadBalancing/simple-backend-
delayed.yaml
kubectl delete destinationrule simple-backend-dr
```

Appliquer la configuration du timeout

```
kubectl apply -f
TP_TP_Data/2_IstioInAction/2.3_Resilience/2.3.2_TimeoutRetry/simple-backend-vs-
timeout.yaml
```

Visualiser les réponses et les erreurs de

```
for in in {1..10}; do time curl -s \
-H "Host: simple-web.formation.io" $INGRESS_HOST:$INGRESS_PORT \
| jq .code; printf "\n"; done
```

Retry

Désactiver le retry automatique :

```
istioctl install --set profile=demo \
```

```
--set meshConfig.defaultHttpRetryPolicy.attempts=0
```

Déployer une version du service ***simple-backend*** qui échoue à 75% (503)

```
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.2_TimeoutRetry/simple-backend-
periodic-failure-408.yaml
```

Visualiser le résultat avec :

```
for in in {1..10}; do curl -s \
-H "Host: simple-web.formation.io" $INGRESS_HOST:$INGRESS_PORT \
| jq .code; printf "\n"; done
```

Autoriser le ré-essai avec

```
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.2_TimeoutRetry/simple-backend-enable-
retry.yaml
```

Retenter les requêtes, vous pouvez également faire un essai avec simple-backend-periodic-failure-500.yaml

2.3.3 Circuit-breaker

Faire en sorte qu'il n'y ait plus qu'un seul pod pour ***simple-backend***

```
kubectl scale deploy/simple-backend-2 –replicas=0
```

Puis déployer la version qui ajoute 1 seconde de délai

```
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.1_ClientLoadBalancing/simple-backend-
delayed.yaml
```

Et enfin supprimer les *DestinationRule*

```
kubectl delete destinationrule –all
```

Reprendre le script JMeter et configurer le pour avoir une connexion effectuant 1 requête par seconde

Limitation pool de connexions et de requêtes

Introduire des limites sur les connexions et les requêtes :

```
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.3_CircuitBreaker/simple-backend-dr-
conn-limit.yaml
```

Ré-exécuter le test

Puis passer le nombre de connexions à 2. Des erreurs 500 doivent apparaître

Pour étendre les statistiques exposés par Istio, notamment les statistique de circuit-breaker

```
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.3_CircuitBreaker/simple-web-stats-
incl.yaml
```

Puis réinitialiser les compteurs avec :

```
kubectl exec -it deploy/simple-web -c istio-proxy \
-- curl -X POST localhost :15000/reset_counters
```

Réexécuter les tests et obtenir les statistiques avec :

```
kubectl exec -it deploy/simple-web -c istio-proxy \
> -- curl localhost:15000/stats | grep simple-backend | grep overflow
```

En particulier :

- ***upstream_cx_overflow*** : Trop de requêtes en //
- ***upstream_rq_pending_overflow*** : Trop de requêtes en attente

Ejection

Réinitialiser le système :

```
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.1_ClientLoadBalancing/simple-
backend.yaml
kubectl delete destinationrule –all
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.3_CircuitBreaker/simple-web-stats-
incl.yaml
```

Désactivation des retry

```
istioctl install --set profile=demo \
--set meshConfig.defaultHttpRetryPolicy.attempts=0
kubectl delete vs simple-backend-vs
```

Introduction d’erreur 500 dans le backend

Exécuter les test JMeter et constater les erreurs

Appliquer la *DestinationRule* définissant les conditions d’éjection :

```
kubectl apply -f
TP_Data/2_IstioInAction/2.3_Resilience/2.3.3_CircuitBreaker/simple-backend-dr-
outlier-5s.yaml
```

Vérifier les améliorations

Accéder aux statistiques d’éjections avec :

```
kubectl exec -it deploy/simple-web -c istio-proxy -- \
```

```
curl localhost:15000/stats | grep simple-backend | grep outlier
```

Finalement, restaurer les réessais et vérifier que le script s’exécute sans erreur.

2.4 Observabilité

Supprimer les add-ons de la démo (Prometheus, Grafana, Kiali)

```
kubectl delete -f $ISTIO_HOME/samples/addons/
```

Supprimer les restes des ateliers précédents :

```
kubectl config set-context $(kubectl config current-context) \
--namespace=formation
kubectl delete virtualservice,deployment,service,\
destinationrule,gateway --all
```

Restaurer une architecture

```
kubectl apply -f TP_Data/1_Demarrer/1.2/catalog.yaml
kubectl apply -f TP_Data/1_Demarrer/1.2/webapp.yaml
kubectl apply -f TP_Data/2_IstioInAction/2.4_Observability/2.4.1_Explore/webapp-catalog-gw-vs.yaml
```

Vérifier avec :

```
curl -H "Host: webapp.formation.io"
http:/./$INGRESS_HOST:$INGRESS_PORT/api/catalog
```

2.4.1 Explorer les statistiques Data Plane

Visualiser les métriques avec :

```
kubectl exec -it deploy/webapp -c istio-proxy \
-- curl localhost:15000/stats
```

Les métriques contiennent des informations sur le proxy et sa connexion à la configuration (nombre de listener mis à jour par exemple) mais aussi des informations sur les requêtes et les réponses.

En particulier, regarder ***istio_requests_total*** en fonction des appels *curl*

Appliquer une configuration personnalisée pour *webapp*

```
kubectl apply -f TP_Data/2_IstioInAction/2.4_Observability/2.4.1_Explore/webapp-deployment-stats-inclusion.yaml
```

Effectuer quelques requêtes

Puis accéder au statistiques :

```
kubectl exec -it deploy/webapp -c istio-proxy \
-- curl localhost:15000/stats | grep catalo
```

Visualiser enfin les information que *webapp* possède sur *catalog*

```
kubectl exec -it deploy/webapp -c istio-proxy \
-- curl localhost:15000/clusters | grep catalog
```

2.4.2 Intégration *Prometheus*

Installer ***kube-prometheus***

```
helm repo add prometheus-community \
https://prometheus-community.github.io/helm-charts
helm repo update
kubectl create ns prometheus
helm install prom prometheus-community/kube-prometheus-stack \
--version 13.13.1 -n prometheus -f
TP_Data/2_IstioInAction/2.4_Observability/2.4.2_Prometheus/prom-values.yaml --
no-hooks
```

Vérifier avec :

```
kubectl get po -n prometheus
```

Configurer ensuite Prometheus pour récupérer les métriques d'Istio en utilisant les ressources ***ServiceMonitor*** et ***PodMonitor*** de Prometheus Operator

```
kubectl apply -f
TP_Data/2_IstioInAction/2.4_Observability/2.4.2_Prometheus/service-monitor-
cp.yaml
```

Exposer le service Prometheus via un port-forward

```
kubectl -n prometheus port-forward \
statefulset/prometheus-prom-kube-prometheus-stack-prometheus 9090
```

Vérifier le scrapping de métriques liés aux services, par exemple ***pilot_xds***

Appliquer le scrapping au niveau du container *istio-proxy* :

```
kubectl apply -f TP_Data/2_IstioInAction/2.4_Observability/2.4.2_Prometheus/pod-
monitor-dp.yaml
```

Générer de la charge :

```
for i in {1..100}; do curl http://$INGRESS_HOST:$INGRESS_PORT/api/catalog -H \
"Host: webapp.formation.io"; sleep .5s; done
```

Visualiser la métrique : ***istio_requests_total***

2.4.3 Visualisation *Grafana*

Exposer le service *Grafana* via un port-forward :

Se logger avec : ***admin/prom-operator***

Importer ensuite les tableaux de bord Istio

```
cd TP_Data/2_IstioInAction/2.4_Observability/2.4.3_Grafana/
```

```
kubectl -n prometheus create cm istio-dashboards \
--from-file=pilot-dashboard.json=dashboards/pilot-dashboard.json \
--from-file=istio-workload-dashboard.json=dashboards/\
istio-workload-dashboard.json \
--from-file=istio-service-dashboard.json=dashboards/\
istio-service-dashboard.json \
--from-file=istio-performance-dashboard.json=dashboards/\
istio-performance-dashboard.json \
--from-file=istio-mesh-dashboard.json=dashboards/\
istio-mesh-dashboard.json \
--from-file=istio-extension-dashboard.json=dashboards/\
istio-extension-dashboard.json
```

La *configmap* créée doit être labelisée afin que Grafana les récupère :

```
kubectl label -n prometheus cm istio-dashboards grafana_dashboard=1
```

Réactualiser Grafana, les tableaux de bord Istio doivent être présents

2.4.4 Tracing distribué

Dans le cadre de cette formation on utilise la distribution de démo d'Istio

```
kubectl apply -f $ISTIO_HOME/samples/addons/jaeger.yaml
```

Vérifier avec :

```
kubectl get pod -n istio-system
kubectl get svc -n istio-system
```

Configuration Istio pour activer Jaeger (Zipkin compatible)

```
istioctl install -y -f
P_Data/2_IstioInAction/2.4_Observability/2.4.4_Jaeger/install-istio-tracing-
zipkin.yaml
```

Afin de visualiser les entêtes, on déploie le service externe ***httpbin***

```
kubectl apply -f TP_Data/2_IstioInAction/2.4_Observability/2.4.4_Jaeger/thin-
httpbin-virtualservice.yaml
```

Puis :

```
curl -H "Host: httpbin.formation.io" http://$INGRESS_HOST:$INGRESS_PORT/headers
```

Repérer les entêtes Zipkin

Visualisation dans Jaeger

Exposer l'UI Jaeger via :

```
istioctl dashboard jaeger –browser=false
```

Exécuter une requête :

```
curl -H "Host: webapp.formation.io"
http://$INGRESS_HOST:$INGRESS_PORT/api/catalog
```

2.4.5 Visualisation avec Kiali

```
kubectl create ns kiali-operator
helm install \
--set cr.create=true \
--set cr.namespace=istio-system \
--namespace kiali-operator \
--repo https://kiali.org/helm-charts \
--version 1.40.1 \
kiali-operator \
kiali-operator
```

Vérifier avec :

```
kubectl get po -n kiali-operator
```

Exposer ensuite le service *Kiali* via du port-forward :

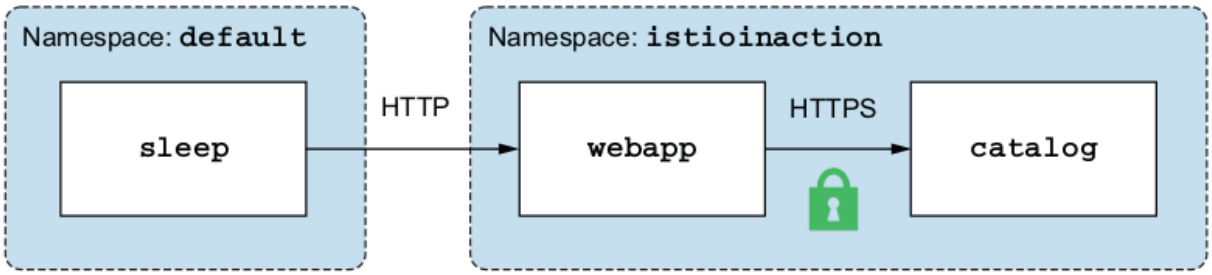
```
kubectl -n istio-system port-forward deploy/kiali 20001
```

2.5 Sécurité

Réinitialiser le service mesh :

```
kubectl config set-context $(kubectl config current-context) \
--namespace=formation
kubectl delete virtualservice,deployment,service,\
destinationrule,gateway –all
```

Pour cet atelier nous introduisons un service legacy qui n'a pas de proxy injecté :



Pour installer les services :

```
kubectl label namespace formation istio-injection=enabled
kubectl apply -f TP_Data/1_Demarrer/1.2/webapp.yaml
kubectl apply -f TP_Data/1_Demarrer/1.2/catalog.yaml
kubectl apply -f
./TP_Data/2_IstioInAction/2.4_Observability/2.4.1_Explore/webapp-catalog-gw-
vs.yaml
kubectl apply -f
./TP_Data/2_IstioInAction/2.5_Security/2.5.1_Securisation/sleep.yaml  -n default
```

2.5.1 Configuration *mTLS*

Vérifier qu'il est possible d'exécuter une requête en clair à partir du pod sleep :

```
kubectl -n default exec deploy/sleep -c sleep -- \
curl -s webapp.formation/api/catalog \
-o /dev/null -w "%{http_code}"
```

Changer la configuration par défaut pour n'autoriser que *mTLS* :

```
kubectl apply -f TP_Data/2_IstioInAction/2.5_Security/2.5.1_mTLS/meshwide-
strict-peer-authn.yaml
```

Vérifier que la requête précédente n'est plus permise

Changer la configuration afin que la workload *webapp* (mais seulement elle) soit accessible de *sleep*

```
kubectl apply -f TP_Data/2_IstioInAction/2.5_Security/2.5.1_mTLS/workload-
permissive-peer-authn.yaml
```

Vérifier que la requête est de nouveau permise

Vérifier ensuite le certificat. (Il doit contenir dune URI SPIFFE)

```
kubectl -n formation exec deploy/webapp -c istio-proxy \
-- openssl s_client -showcerts \
-connect catalog.formation.svc.cluster.local:80 \
-CAfile /var/run/secrets/istio/root-cert.pem | \
openssl x509 -in /dev/stdin -text -noout
```

2.5.2 Autorisation

Commencer par appliquer un ***deny catch-all***

```
kubectl apply -f TP_Data/2_IstioInAction/2.5_Security/2.5.2_Autorisation/policy-deny-all-mesh.yaml
```

Tenter la requête

```
kubectl -n default exec deploy/sleep -c sleep -- \
curl -sSL webapp.formation/api/catalog
```

Autoriser un service non identifié

```
kubectl apply -f TP_Data/2_IstioInAction/2.5_Security/2.5.2_Autorisation/allow-unauthenticated-view-default-ns.yaml
```

Ré-essayer la requête. Qu’observez vous ?

Autoriser en fonction de l’identité du compte de service

Ajouter une règle afin que la requête soit possible

2.5.3 Utilisateur final et JWT

Réinitialiser l’environnement :

```
kubectl config set-context $(kubectl config current-context) \
--namespace=formation
kubectl delete virtualservice,deployment,service,\
destinationrule,gateway,peerauthentication,authorizationpolicy --all
kubectl delete peerauthentication,authorizationpolicy \
-n istio-system –all
```

Déployer

```
kubectl apply -f TP_Data/1_Demarrer/1.2/webapp.yaml
kubectl apply -f TP_Data/1_Demarrer/1.2/catalog.yaml
kubectl apply -f TP_Data/2_IstioInAction/2.5_Security/2.5.3_EndUserJWT/ingress-
```

```
gw-for-webapp.yaml
```

Ajouter une ressource ***RequestAuthentication*** incluant la clé JWKS :

```
kubectl apply -f TP_Data/2_IstioInAction/2.5_Security/2.5.3_EndUserJWT/jwt-token-request-authn.yaml
```

Faire une requête avec un jeton valide :

```
USER_TOKEN=$(< TP_Data/2_IstioInAction/2.5_Security/2.5.3_EndUserJWT/user.jwt); curl -H "Host: webapp.formation.io" -H "Authorization: Bearer $USER_TOKEN" -sSl -o /dev/null -w "%{http_code}" $INGRESS_HOST:$INGRESS_PORT/api/catalog
```

Une requête avec un issuer incorrect

```
WRONG_ISSUER=$(< TP_Data/2_IstioInAction/2.5_Security/2.5.3_EndUserJWT/not-configured-issuer.jwt); \
curl -H "Host: webapp.formation.io" \
-H "Authorization: Bearer $WRONG_ISSUER" \
-sSl $INGRESS_HOST:$INGRESS_PORT/api/catalog
```

Une requête sans jeton

```
curl -H "Host: webapp.formation.io" \
-sSl -o /dev/null -w "%{http_code}" $INGRESS_HOST:$INGRESS_PORT/api/catalog
```

Forcer l’utilisation de JWT

```
kubectl apply -f TP_Data/2_IstioInAction/2.5_Security/2.5.3_EndUserJWT/app-gw-requires-jwt.yaml
```

Retenter la requête sans jeton

Ajouter 2 autres règles d’autorisation :

- GET permis pour tous les users
- Tout est permis si on est administrateur

```
kubectl apply -f TP_Data/2_IstioInAction/2.5_Security/2.5.3_EndUserJWT/allow-all-with-jwt-to-webapp.yaml
kubectl apply -f TP_Data/2_IstioInAction/2.5_Security/2.5.3_EndUserJWT/allow-mesh-all-ops-admin.yaml
```

Vérifier avec :

```
USER_TOKEN=$(< TP_Data/2_IstioInAction/2.5_Security/2.5.3_EndUserJWT/user.jwt); curl -H "Host: webapp.formation.io" -H "Authorization: Bearer $USER_TOKEN" -sSl -o /dev/null -w "%{http_code}" $INGRESS_HOST:$INGRESS_PORT/api/catalog
```

et

```
USER_TOKEN=$(< TP_Data/2_IstioInAction/2.5_Security/2.5.3_EndUserJWT/user.jwt); curl -H "Host: webapp.formation.io" -H "Authorization: Bearer $USER_TOKEN" -sSl -o /dev/null -w "%{http_code}" -XPOST $INGRESS_HOST:$INGRESS_PORT/api/catalog --data '{"id": 2, "name": "Shoes", "price": "84.00"}'
```

et

```
ADMIN_TOKEN=$(< TP_Data/2_IstioInAction/2.5_Security/2.5.3_EndUserJWT/admin.jwt); curl -H "Host: webapp.formation.io" -H "Authorization: Bearer $ADMIN_TOKEN" -sSl -o /dev/null -w "%{http_code}" -XPOST $INGRESS_HOST:$INGRESS_PORT/api/catalog --data '{"id": 2, "name": "Shoes", "price": "84.00"}'
```

3. Autres

3.1 Installation

3.1.1 Découplage ControlPlane Gateway avec istioCtl

Désinstaller tout :

```
istioctl uninstall --purge
kubectl delete deployment,service -n istio-system --all
kubectl delete namespace istio-system
```

Vérifier qu'aucun pod ne s'exécute dans l'espace de nom istio-system

Appliquer une ressource IstioOperator qui configure le control plane sans les gateways

```
istioctl install -f
TP_Data/3_Autres/3.1_Installation/3.1.1_DecouplageControlGateway/demo-profile-
without-gateways.yaml
```

Vérifier en regardant les pods de l'espace de nom istio-system

Appliquer une ressource IstioOperator partant du profil empty et installant les gateways

```
istioctl install -f
TP_Data/3_Autres/3.1_Installation/3.1.1_DecouplageControlGateway/ingress-
gateway.yaml
```

Vérifier les pods

3.1.2 Installation avec istio-operator

On peut installer l'opérateur avec :

```
istioctl operator init
```

L'installation précédente peut alors s'effectuer juste en créant les ressources ***IstioOperator*** dans l'espace de nom ***istio-system*** :

```
kubectl apply -f
TP_Data/3_Autres/3.1_Installation/3.1.1_DecouplageControlGateway/demo-profile-
without-gateways.yaml -n istio-system
```

```
kubectl apply -f
TP_Data/3_Autres/3.1_Installation/3.1.1_DecouplageControlGateway/demo-profile-
without-gateways.yaml -n istio-system
```

Mette à jour l'installation du control plane via :

```
kubectl apply -f TP_Data/3_Autres/3.1_Installation/3.1.2_IstioOperator/demo-
```

```
profile-without-gateways-json.yaml -n istio-system
```

3.1.3 Résolution de problèmes

Nettoyer :

```
kubectl config set-context $(kubectl config current-context) \
--namespace=formation
kubectl delete virtualservice,deployment,service,\
destinationrule,gateway,authorizationpolicy,peerauthentication --all
kubectl delete authorizationpolicy,peerauthentication --all -n istio-system
```

Déployer

```
kubectl apply -f TP_Data/1_Demarrer/1.2/catalog.yaml
kubectl apply -f TP_Data/3_Autres/3.2_Problemes/catalog-deployment-v2.yaml
kubectl apply -f TP_Data/3_Autres/3.2_Problemes/catalog-gateway.yaml
kubectl apply -f TP_Data/3_Autres/3.2_Problemes/catalog-virtualservice-subsets-v1-v2.yaml
```

Tester avec :

```
for i in {1..100}; do curl http://$INGRESS_HOST:$INGRESS_PORT/items \
-H "Host: catalog.formation.io" \
-w "\nStatus Code %{http_code}\n"; sleep .5s; done
```

Utiliser les outils pour debugger

Ensuite, exécuter les commandes suivantes :

```
CATALOG_POD=$(kubectl get pods -l version=v2 -n formation -o \
jsonpath={.items..metadata.name} | cut -d ' ' -f1) \
kubectl -n formation exec -c catalog $CATALOG_POD \
-- curl -s -X POST -H "Content-Type: application/json" \
-d '{"active": true, "type": "latency", "volatile": true}' \
$INGRESS_HOST:$INGRESS_PORT:3000/blowup
kubectl patch vs catalog-v1-v2 -n formation --type json \
-p '[{"op": "add", "path": "/spec/http/0/timeout", "value": "0.5s"}]'
```

Générer de la charge avec

```
for i in {1..9999}; do curl http://$INGRESS_HOST:$INGRESS_PORT/items \
-H "Host: catalog.formation.io" \
-w "\nStatus Code %{http_code}\n"; sleep 1s; done
```

Visualisez le tableau de bord Grafana et essayer d’identifier dans les logs d’accès d’Envoy les requêtes posant problèmes

3.1.4 Monitoring de istiod

Nettoyer

```
kubectl config set-context $(kubectl config current-context) \
--namespace=formation
kubectl delete virtualservice,deployment,service,\
destinationrule,gateway --all
```

Déployer

```
kubectl apply -f TP_Data/1_Demarrer/1.2/catalog.yaml
kubectl apply -f TP_Data/3_Autres/3.3_Istiod/catalog-gateway.yaml
kubectl apply -f TP_Data/3_Autres/3.3_Istiod/catalog-virtualservice.yaml
kubectl apply -f TP_Data/3_Autres/3.3_Istiod/sleep-dummy-workloads.yaml
kubectl -n formation apply -f TP_Data/3_Autres/3.3_Istiod/resources-600.yaml
```

Exécuter un test de performance avec

```
TP_Data/3_Autres/3.3_Istiod//bin/performance-test.sh --reps 10 --delay 2.5 --
gateway ÎNGRESS_HOST:$INGRESS_PORT
```

Pendant le test visualiser les tableaux de bord Grafana