Create our Persistent Volume Definition for the database (db-pv.yaml):

apiVersion: v1

kind: PersistentVolume

metadata:

name: db-pv-volume

labels:

type: local

spec:

storageClassName: manual

capacity:

storage: 512Mi

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Retain

hostPath:

path: "/home/gavin/data/db"

type: DirectoryOrCreate

Create our Persistent Volume Claim that will map to the above (dv-pv-claim.yaml):

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: task-pv-claim

spec:

storageClassName: manual

accessModes:

- ReadWriteOnce

selector:

matchLabels:

type: local

resources:

requests:

storage: 256Mi

Add the Claim/Volume to the deployment (qservice-dep.yaml):

apiVersion: apps/v1

kind: Deployment

metadata:

creationTimestamp: null

labels:

app: qservice-dep

name: qservice-dep

spec:

replicas: 1

selector:

matchLabels:

app: qservice-app

strategy: {}

template:

metadata:

creationTimestamp: null

labels:

app: qservice-app

spec:

volumes:

- name: db-pv-store

persistentVolumeClaim:

claimName: task-pv-claim

containers:

- image: gavinpollitt/qservice

name: qservice

- image: buildo/h2database

name: qservice-store

volumeMounts:

- mountPath: "/h2-data"

name: db-pv-store

Add the ConfigMap to contain the application.yaml for the service (config-app.yaml):

apiVersion: v1

kind: ConfigMap

metadata:

name: qservice-config

data:

application.yml: |

spring:

datasource:

driverClassName: org.h2.Driver

url: jdbc:h2:tcp://localhost:1521/gav

username:

password:

jpa:

database-platform: org.hibernate.dialect.H2Dialect

hibernate:

ddl-auto: update

properties:

hibernate:

format\_sql: false

show\_sql: true

use\_sql\_comments: false

server:

port: 8081

queue:

maxEntries: 6

timeout: 5

Update the deployment with (qservice-dep.yaml) the config map mount:

apiVersion: apps/v1

kind: Deployment

metadata:

creationTimestamp: null

labels:

app: qservice-dep

name: qservice-dep

spec:

replicas: 1

selector:

matchLabels:

app: qservice-app

strategy: {}

template:

metadata:

creationTimestamp: null

labels:

app: qservice-app

spec:

volumes:

- name: db-pv-store

persistentVolumeClaim:

claimName: task-pv-claim

**- name: config-volume**

**configMap:**

**name: qservice-config**

containers:

- image: gavinpollitt/qservice

name: qservice

**volumeMounts:**

**- name: config-volume**

**mountPath: /home/config**

- image: buildo/h2database

name: qservice-store

volumeMounts:

- mountPath: "/h2-data"

name: db-pv-store

This can now be run and logged into with:

kubectl exec -it <pod> -c qservice – sh

Now create the service to expose the qservice – nodePort initially:

kubectl expose deploy qservice-dep --port 8081 --type NodePort

Check things are accessible:

curl -i -X POST -d @cc.json -H "Content-Type: application/json" http://localhost:<node port>/queueService/add

*cc Json = {"name":"GavWebCo2","description":"The final description", "userName":"gavlad"}*

Update the service to return to a ClusterIP

kubectl get svc qservice-dep -o yaml (reapply without NodePort and change type to ClusterIP):

Set node affinity, so that we don’t lose volume on restart:

Adjust the deployment:

…

spec:

volumes:

- name: db-pv-store

persistentVolumeClaim:

claimName: task-pv-claim

- name: config-volume

configMap:

name: qservice-config

containers:

- image: gavinpollitt/qservice

name: qservice

volumeMounts:

- name: config-volume

mountPath: /home/config

- image: buildo/h2database

name: qservice-store

volumeMounts:

- mountPath: "/h2-data"

name: db-pv-store

**nodeSelector:**

**type: data**

Set the label on the node:

*kubectl label node docker-desktop type=data*

Now, reapply the deployment.

Now let’s create a network policy to (np.yaml):

* **Only allow access on port 8081 for pods with label – ‘app:qservice-app’**
* **Only allow access from the client namespace**
* **Only allow access from pods with the *role:qclient* label**

Create the namespace and label it first:

*kubectl create namespace client*

*kubectl label ns client source=frontend*

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: q-network-policy

namespace: default

spec:

podSelector:

matchLabels:

app: qservice-app

policyTypes:

- Ingress

ingress:

- from:

- namespaceSelector:

matchLabels:

source: frontEnd

- podSelector:

matchLabels:

role: qClient

ports:

- protocol: TCP

port: 8081

Now let’s create our client controller.

Once again…an application ConfigMap:

apiVersion: v1

kind: ConfigMap

metadata:

name: qclient-config

namespace: client

data:

application.yml: |

server:

port: 8080

qservice:

url: http://qservice-dep:8081/queueService

**…and the deployment:**

apiVersion: apps/v1

kind: Deployment

metadata:

creationTimestamp: null

labels:

app: qclient-dep

name: qclient-dep

namespace: client

spec:

replicas: 3

selector:

matchLabels:

app: qclient-app

template:

metadata:

creationTimestamp: null

labels:

app: qclient-app

spec:

volumes:

- name: config-volume

configMap:

name: qclient-config

containers:

- image: gavinpollitt/qclient

name: qclient

volumeMounts:

- name: config-volume

mountPath: /home/config

Now expose as a NodePort service for ‘external’ consumption:

kubectl expose deploy qclient-dep --port 8080 --type NodePort -n client

Ok, let’s consolidate all of our logs into one simple log…(in the kube-system namespace)

Add sidecar container to client deployment to push logs to its stdout (qclient-dep.yaml):

spec:

volumes:

- name: config-volume

configMap:

name: qclient-config

**- name: log-volume**

**emptyDir: {}**

containers:

- image: gavinpollitt/qclient

name: qclient

volumeMounts:

- name: config-volume

mountPath: /home/config

**- name: log-volume**

**mountPath: /logs**

**- image: busybox:1.28**

**name: qlogger**

**args: [/bin/sh, -c, 'tail -n+1 -f ./logs/qclient.log']**

**volumeMounts:**

**- name: log-volume**

**mountPath: ./logs**

Let’s add a dummy service that acts as our consolidated log consumer (logreceiver-pod.yaml):

Firstly, give node a role to pin the consumer pod**: kubectl label node docker-desktop "role=ops"**

*Now create the consumer pod pinned to this node:*

apiVersion: v1

kind: Pod

metadata:

creationTimestamp: null

labels:

run: logreceiver-pod

name: logreceiver-pod

namespace: kube-system

spec:

containers:

- image: mendhak/http-https-echo

name: logreceiver-pod

dnsPolicy: ClusterFirst

restartPolicy: Always

nodeSelector:

role: ops

*OK – And expose through a service :*

apiVersion: v1

kind: Service

metadata:

labels:

app: logreceiver

name: logreceiver

namespace: kube-system

spec:

ports:

- port: 8080

protocol: TCP

targetPort: 80

selector:

run: logreceiver-pod

sessionAffinity: None

type: ClusterIP

status:

loadBalancer: {}

Now FluentD DeamonPod that will ‘mine’ the node logs for our ‘shared’ client service logs (fluent-ds.yaml):

Firstly, the DeamonPod must be allowed to talk to the Kubernetes API Controller, so a service account is created (still in the kube-system namespace):

apiVersion: v1

kind: ServiceAccount

metadata:

name: fluentd

namespace: kube-system

labels:

k8s-app: fluentd-logging

version: v1

Then, create the cluster role that will give the FluentD service its capability:

apiVersion: rbac.authorization.k8s.io/v1beta1

kind: ClusterRole

metadata:

name: fluentd

namespace: kube-system

rules:

- apiGroups:

- ""

resources:

- pods

- namespaces

verbs:

- get

- list

- watch

Then, ‘bind’ the above two together:

apiVersion: rbac.authorization.k8s.io/v1beta1

kind: ClusterRoleBinding

metadata:

name: fluentd

roleRef:

kind: ClusterRole

name: fluentd

apiGroup: rbac.authorization.k8s.io

subjects:

- kind: ServiceAccount

name: fluentd

namespace: kube-system

Finally, the DaemonSet itself can be defined:

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd

namespace: kube-system

labels:

k8s-app: fluentd-logging

version: v1

spec:

selector:

matchLabels:

k8s-app: fluentd-logging

template:

metadata:

labels:

k8s-app: fluentd-logging

version: v1

spec:

serviceAccount: fluentd

serviceAccountName: fluentd

tolerations:

- key: node-role.kubernetes.io/master

effect: NoSchedule

containers:

- name: fluentd

image: gavinpollitt/kubfluentd:1.2

resources:

limits:

memory: 200Mi

requests:

cpu: 100m

memory: 200Mi

volumeMounts:

- name: varlog

mountPath: /var/log

- name: varlibdockercontainers

mountPath: /var/lib/docker/containers

readOnly: true

- name: config-volume

mountPath: /fluentd/etc

terminationGracePeriodSeconds: 30

volumes:

- name: varlog

hostPath:

path: /var/log

- name: varlibdockercontainers

hostPath:

path: /var/lib/docker/containers

- name: config-volume

configMap:

name: fluentd-config

The two hostpath volumes expose the log directories of the node on which the DaemonSet runs, which will contains the ‘stdout’ feed from the ‘qlogger’ sidecar containers of the queue client application. The config map volume contains the configuration required by fluentd to ‘find’ the appropriate logs and ‘ship’ to the ‘logreceiver’ service.

The ConfigMap for the Fluentd config is defined in the file – fluent-config.yaml:

apiVersion: v1

kind: ConfigMap

metadata:

name: fluentd-config

namespace: kube-system

data:

fluent.conf: |

<source>

@type tail

@id in\_tail\_container\_logs

path /var/log/containers/qclient-dep\*qlogger\*.log

pos\_file /var/log/fluentd-containers.log.pos

tag kubernetes.\*

read\_from\_head true

<parse>

@type json

time\_format %Y-%m-%dT%H:%M:%S.%NZ

</parse>

</source>

<match \*\*>

@type http

endpoint http://logreceiver:8080/api

open\_timeout 2

<format>

@type json

</format>

<buffer>

flush\_interval 10s

</buffer>

</match>

Metric Service:

* wget <https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.3.7/components.yaml>
* Ensure args in ‘metrics-server’ deployment look like if it’s struggling:

args:

- --cert-dir=/tmp

- --secure-port=4443

- --**kubelet-insecure-tls=true**

**- --kubelet-preferred-address-types=InternalIP**

* It may take a while to start

Few checks:

* Check service app is top of the list in default namespace : **kubectl top pod --no-headers --sort-by='memory' | head -n1**
* Fluentd should be top in ‘kube-system’: **kubectl top pod --no-headers --sort-by='memory' -n kube-system | head -n1**
* Check the 3 pods for ‘qservice-dep’: **kubectl get pods -n client --selector="app=qclient-app"**
* Scale the deployment if required:

kubectl scale deploy qclient-dep --replicas=4 -n client –record

kubectl rollout history deploy qclient-dep -n client

* Ensure that killing the q service pods, causing an auto-restart and retains data:
  + Add some companies: **curl -v -X POST -d @cc.json -H "Content-Type: application/json" http://localhost:<NodePort>/queue/addCompany**
  + Kill the service **pod** from the default namespace: **kubectl delete pod qservice-dep-<rest>**
  + Ensure new pod is created (on same node): **kubectl get pods -o wide**
  + Get some entries off the queue:

**curl -H "Content-Type: application/json" -X GET** [**http://localhost:31327/queue/getCompany?userName="carol**](http://localhost:31327/queue/getCompany?userName="carol)**"**

**curl -H "Content-Type: application/json" -X GET http://localhost:31327/queue/getCompanies/3?userName="noddy"**

* Next

Let’s now set up our Ingress (ingress.yaml):

Start as basic http – don’t forget, just having this file doesn’t do anything ingress related until backed up by a controller.

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

annotations:

kubernetes.io/ingress.class: "nginx"

nginx.ingress.kubernetes.io/rewrite-target: / # write same as incoming path

nginx.ingress.kubernetes.io/use-regex: true

name: client-ingress

namespace: client

spec:

rules:

- http:

paths:

- path: /queue

#pathType: Prefix

backend:

serviceName: qclient-dep

servicePort: 8080

We then need to deploy the actual controller – in this case Nginx (note this is for docker-desktop) (nginx-controller-deploy.yaml):

Apply the yaml and run the following to determine completion of set-up:

kubectl wait --namespace ingress-nginx \

--for=condition=ready pod \

--selector=app.kubernetes.io/component=controller \

--timeout=120s

To see the running version:

POD\_NAMESPACE=ingress-nginx

POD\_NAME=$(kubectl get pods -n $POD\_NAMESPACE -l app.kubernetes.io/name=ingress-nginx --field-selector=status.phase=Running -o jsonpath='{.items[0].metadata.name}')

kubectl exec -it $POD\_NAME -n $POD\_NAMESPACE -- /nginx-ingress-controller –version

Do a few checks:

curl -v -X POST -d @cc.json -H "Content-Type: application/json" <http://localhost:80/queue/addCompany>

curl -X GET <http://localhost:80/queue/getCompany?userName="gav>"

Now, in readiness for TLS introduction, create a specific hostname to the ingress (and add to /etc/hosts, if required):

spec:

rules:

**- host: gav.wsl2**

http:

paths:

- path: /queue

#pathType: Prefix

backend:

serviceName: qclient-dep

servicePort: 8080

Test with revised URL: **curl -v -X POST -d @cc.json -H "Content-Type: application/json"** [**http://gav.wsl2:80/queue/addCompany**](http://gav.wsl2:80/queue/addCompany)

**NOTE: May need to comment out mapping and regex from annotations in Ingress file depending on environment.**

Let’s secure our Ingress endpoint for encrypted traffic:

Firstly, generate a key and cert based on the new hostname:

export KEY\_FILE=./queue.key

export CERT\_FILE=./queue.cert

export HOST=gav.wsl2

export CERT\_NAME=gavq

openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ${KEY\_FILE} -out ${CERT\_FILE} -subj "/CN=${HOST}/O=${HOST}"

Create the secret to hold this information in the same namespace as the Ingress rule:

*kubectl create secret tls ${CERT\_NAME} --key ${KEY\_FILE} --cert ${CERT\_FILE} -n client*

Now, revise the Ingress Rules to provide TLS information:

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

annotations:

kubernetes.io/ingress.class: "nginx"

nginx.ingress.kubernetes.io/rewrite-target: /

nginx.ingress.kubernetes.io/use-regex: true

name: client-ingress

namespace: client

spec:

**tls:**

**- hosts:**

**- gav.wsl2**

**secretName: gavq**

rules:

- host: gav.wsl2

http:

paths:

- path: /queue

#pathType: Prefix

backend:

serviceName: qclient-dep

servicePort: 8080

Test it:

curl -X GET <http://gav.wsl2:80/queue/getCompany?userName="gav>"

?

*<html>*

*<head><title>308 Permanent Redirect</title></head>*

*<body>*

*<center><h1>308 Permanent Redirect</h1></center>*

*<hr><center>nginx/1.19.2</center>*

*</body>*

*</html>*

curl -X GET <https://gav.wsl2/queue/getCompany?userName="gav>"



curl: (60) SSL certificate problem: self signed certificate

*More details here: https://curl.haxx.se/docs/sslcerts.html*

*curl failed to verify the legitimacy of the server and therefore could not*

*establish a secure connection to it. To learn more about this situation and*

*how to fix it, please visit the web page mentioned above.*

Now, add the certificate file to the curl command:

curl --cacert <path>/queue.cert -X GET https://gav.wsl2/queue/getCompany?userName="

gav"

*{"userName":"gav","id":51,"name":"GavWebCo2","description":"The final description","createdDate":"2020-09-24"}*

Backup our etcd cluster persistence data:

Open up **/etc/kubernetes/manifests/etcd.yaml** and get the following:

--cert-file

--key-file

--trusted-ca-file

Save the cluster:

Do the following in sudo -i:

**ETCDCTL\_API=3 etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key snapshot save \**

**/home/gavin/mock/gavclust.save**

Delete an unrequired pod.

then restore:

Do the following in sudo -i:

**ETCDCTL\_API=3 etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key snapshot restore /home/gavin/mock/gavclust.save**

Now, upgrade the cluster to the latest version:

Get the latest version (sudo):

**apt update**

**apt-cache madison kubeadm**

e.g. 1.19.2-00

Update the control plane – controller (sudo):

**apt-get update && \**

**apt-get install -y --allow-change-held-packages kubeadm=1.19.2-00**

version: **kubeadm version**

Drain the control plane node (non-sudo):

**kubectl drain controller --ignore-daemonsets**

Update the plan:

**sudo kubeadm upgrade plan**

Using the command output, do the upgrade:

**sudo kubeadm upgrade apply v1.19.2**

**Update the CNI, if required**

Uncordon:

**kubectl uncordon controller**

Upgrade kubelet and kubectl (sudo):

**apt-get update && \**

**apt-get install -y --allow-change-held-packages kubelet=1.19.2-00 kubectl=1.19.2-00**

Restart the Kubelet:

**sudo systemctl daemon-reload**

**sudo systemctl restart kubelet**

**Now do the same for the nodes/workers:**

**Note: The kubectl commands will need to be done on the controller.**

**Upgrade kubeadm**

When draining***: kubectl drain worker-x --ignore-daemonsets --delete-local-data –force***

**sudo kubeadm upgrade node**

***Update Kubelet and Kubectl as in control plane***

***Restart kubelet and uncordon nodes***

On ARM (Pi) with K3S:

Some initial set-up:

Key thing on Ubuntu 20 for PI is to enable cgroups properly:

Amend file: **/boot/firmware/cmdline.txt**

to contain: **net.ifnames=0 dwc\_otg.lpm\_enable=0 console=serial0,115200 cgroup\_enable=cpuset cgroup\_enable=memory cgroup\_memory=1 console=tty1 root=LABEL=writable rootfstype=ext4 elevator=deadline rootwait fixrtc**

Install docker.ce and enable service (if want docker, rather than containerd) - https://docs.docker.com/engine/install/ubuntu/

Install K3s: **curl -sfL https://get.k3s.io | sh -s - --docker --disable traefik** (to use docker in lieu of containerd and prevent traffic Ingress Controller); https://ikarus.sg/kubernetes-with-k3s/; https://rancher.com/docs/k3s/latest/en/

NOTE: When connecting agent, use IP of master as opposed to DNS as the DNS doesn't seem to resolve.

**App Notes:**

Had to tweak the qservice app to ensure that the config was loaded first – hence to ARM feature created.

Had to adjust the qservice container to use the arm version of the gavinpollitt/qservice container.

To allow network policy to work, changed default Flannel CNI to Calico – upgraded using instructions from: [**https://stackoverflow.com/questions/59054628/kubernetes-replace-flannel-with-calic**o](https://stackoverflow.com/questions/59054628/kubernetes-replace-flannel-with-calico)

Had to adjust the qclient container to use the arm version of the gavinpollitt/qclient container.

Also had to add the role label to client due to tightened network policy

**Monitor Notes:**

Change the logreceiver image to an ARM compatible one – wiremock in this case!

**Ingress Notes:**

K3s uses the Klipper load balancer by default. This creates a network route from a host port (in this case 80 and 443) to one of its own DaemonSet pods, which, in turn, forwards to the Nginx Ingress Controller. To see the host IP that’s been used as the external IP:

**kubectl get svc ingress-nginx-controller -n ingress-nginx**

**DNS Pain in the But!**

Through the use of CoreDNS, the *etc/hosts* file isn’t referenced in the PODS and for the host: **gav-wsl2.poll.com**, the DNS ‘chain’ doesn’t contain this entry, so, ultimately, the site ‘poll.com’ on the internet is referenced. This meant that the local UAA service was not used to auth.

To correct this, the host had to be added to the CoreDNS hosts set to ensure that the correct IP was ‘released’ by DNS when referenced in the PODS:

*kubectl -n kube-system edit configmap/coredns*

and ensure that it’s updated with:

...

*}*

*NodeHosts: |*

*192.168.1.25 control-pi*

*192.168.1.30 worker0-pi*

*192.168.1.31 worker1-pi*

***192.168.1.25 gav-wsl2.poll.com***

*kind: ConfigMap*

*...*

Recreate the ingress-nginx-controller pod and test with a lookup from the pod:

*nslookup gav-wsl2.poll.com*

should get correct IP instead of 90. something.

**Add the Config Server**

Creates a running instance of Spring Cloud Config server using a local (to the controller node) git repo to hold the config. Mounts locally to pod at /config as identified by the environment variable. (spring-conf-dep.yaml)

apiVersion: apps/v1

kind: Deployment

metadata:

creationTimestamp: null

labels:

app: spring-conf-dep

name: spring-conf-dep

namespace: config

spec:

replicas: 1

selector:

matchLabels:

app: spring-conf-dep

strategy: {}

template:

metadata:

creationTimestamp: null

labels:

app: spring-conf-dep

spec:

volumes:

- name: git-repo-vol

hostPath:

path: "/home/ubuntu/git/repo"

type: Directory

containers:

- image: gavinpollitt/spring-config-arm64:1.0

name: spring-cloud-config-server

env:

- name: SPRING\_CLOUD\_CONFIG\_SERVER\_GIT\_URI

value: file:/config

volumeMounts:

- mountPath: "/config"

name: git-repo-vol

ports:

- containerPort: 8888

nodeSelector:

type: data

status: {}

Expose as a service to be consumed by the application pods (**spring-conf-svc.yaml**):

apiVersion: v1

kind: Service

metadata:

creationTimestamp: null

labels:

app: spring-conf-dep

name: spring-conf-dep

namespace: config

spec:

ports:

- port: 8888

protocol: TCP

targetPort: 8888

selector:

app: spring-conf-dep

status:

loadBalancer: {}

~

To check things are working, log into a busybox pod and issue:

**wget http://spring-conf-dep.config:8888/qserver-pi/dev**

**Now adjust the service deployment to use the remote config:**

Adjust the service config to include the bootstrap configuration too (config-app.yaml) – this will be the base configuration before remote adjustments are added:

apiVersion: v1

kind: ConfigMap

metadata:

name: qservice-config

data:

application.yml: |

spring:

jpa:

database-platform: org.hibernate.dialect.H2Dialect

hibernate:

ddl-auto: update

properties:

hibernate:

format\_sql: false

show\_sql: true

use\_sql\_comments: false

queue:

maxEntries: 3

timeout: 5

server:

port: 8081

management:

security:

enabled: false

bootstrap.yml: |

spring:

application:

name: qserver-pi

cloud:

config:

uri: http://spring-conf-dep.config:8888

enabled: false

**The files loaded into the GIT repository consumed by Spring Config pod are:**

The base config in: qservice-pi.yaml:

spring:

jpa:

datasource:

driverClassName: org.h2.Driver

url: jdbc:h2:tcp://localhost:1521/gav

username:

password:

queue:

maxEntries: 3

timeout: 5

The specific dev config (which the server app is running under) in: qservice-pi-dev.yaml:

queue:

maxEntries: 7

This results in a superset of the config:

base (from configmap) + root (from config) + dev from configuration

**Now, add the Zipkin and Sleuth Support**

Zipkin is effectively a metrics view (in-memory initially) of traced routes through the application services. In our case, automated touch-points by Sleuth are:

- The client controller receiving a REST call

- The client controller issuing a server REST call following receipt of the above

- The service controller receiving this REST call

Sleuth adds a trace id amongst other things and can then be viewed in the Zipkin UI:

- [http://gav-wsl2.poll.com](http://gav-wsl2.poll.com/):<NodePort>/zipkin

**Add the the Zipkin deployment (zipkin-dep.yaml):**

apiVersion: apps/v1

kind: Deployment

metadata:

creationTimestamp: null

labels:

app: zipkin-dep

name: zipkin-dep

namespace: config

spec:

replicas: 1

selector:

matchLabels:

app: zipkin-dep

strategy: {}

template:

metadata:

creationTimestamp: null

labels:

app: zipkin-dep

spec:

containers:

- image: gavinpollitt/zipkin:arm-1.0

name: zipkin

ports:

- containerPort: 9411

**nodeSelector:**

**type: data #Once again, pin to the data node**

**...and expose the service:**

apiVersion: v1

kind: Service

metadata:

creationTimestamp: null

labels:

app: zipkin-dep

name: zipkin-dep

namespace: config

spec:

type: NodePort

ports:

- port: 9411

protocol: TCP

targetPort: 9411

selector:

app: zipkin-dep

**To consume this endpoint, the applications need to be *instrumented* to utilise the appropriate hooks into Zipkin:**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

**...and update the spring-config dynamic data to provide the Zipkin endpoint:**

spring:

zipkin:

base-url: http://zipkin-dep.config.svc.cluster.local:9411

sleuth:

sampler:

probability: 1.0 **#ensure the sleuth picks up everything**

**We are also bootstrapping the client now too allowing it to consume from spring-config (config-app.yaml):**

apiVersion: v1

kind: ConfigMap

metadata:

name: qclient-config

namespace: client

data:

application.yml: |

server:

port: 8080

management:

security:

enabled: false

qservice:

url: http://qservice-dep.default.svc.cluster.local:8081/queueService

**bootstrap.yml: |**

**spring:**

**application:**

**name: qclient-pi**

**cloud:**

**config:**

**uri: http://spring-conf-dep.config:8888**

**enabled: false**

**Now, add some health checks for the service:**

We want to ensure that the client is only deemed health once the queue-service is available for consumption. You’d think that this could be done by a livenessProbe against the remote service; however, as these are performed by the Kubelet and the Kubelet doesn’t have access to the cluster DNS, it needs to be done by alternative means. In this case, we use an init container to do the check (qclient-dep.yaml):

...

spec:

replicas: 3

selector:

matchLabels:

app: qclient-app

template:

metadata:

creationTimestamp: null

labels:

app: qclient-app

role: qClient

spec:

volumes:

- name: config-volume

configMap:

name: qclient-config

- name: log-volume

emptyDir: {}

**initContainers:**

**- image: busybox:1.28**

**name: check-server**

**#In lieu of liveness/readiness check not being able to resolve the DNS name (due to being called by Kubelet)**

**command: ['sh', '-c', 'for i in $(seq 1 6); do X=$(wget -q -O - http://qservice-dep.default:8081/actuator/health | grep "UP"); [ ! -z ${X} ] && break; sleep 10; done']**

containers:

…

The command loops until the remote Spring Boot health check says it’s up. If the loop expires, the client will be deemed unhealthy. The actual application containers of the client will only become active following a successful completion of the init container.

A readiness check is also performed against the application container instances themselves to ensure that they are up and running (qclient-dep.yaml):

...

spec:

volumes:

- name: config-volume

configMap:

name: qclient-config

- name: log-volume

emptyDir: {}

initContainers:

- image: busybox:1.28

name: check-server

#In lieu of liveness/readiness check not being able to resolve the DNS name (due to being called by Kubelet)

command: ['sh', '-c', 'for i in $(seq 1 6); do X=$(wget -q -O - http://qservice-dep.default:8081/actuator/health | grep "UP"); [ ! -z ${X} ] && break; sleep 10; done']

containers:

- image: gavinpollitt/qclient:arm-1.3

name: qclient

volumeMounts:

- name: config-volume

mountPath: /home/config

- name: log-volume

mountPath: /logs

**readinessProbe:**

**httpGet:**

**path: /actuator/health**

**port: 8080**

**initialDelaySeconds: 10 #don’t both checking for 10 seconds**

**periodSeconds: 10 #check every 10 seconds**

**timeoutSeconds: 20 #a check fails if it times out.**

…