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"}*

curl -i -X GET <http://localhost:30005/queueService/get>

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=%22carol)**"**

**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](http://localhost:80/queue/getCompany?userName=%22gav)"

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](http://gav.wsl2:80/queue/getCompany?userName=%22gav)"

?

*<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](https://gav.wsl2/queue/getCompany?userName=%22gav)"



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***

**Now adapt to cater for OAuth**

Let’s use Cloudfoundry UAA as our Authorisation/OpenID service.

Let all our OAuth services reside in the ‘security’ namespace (sec-ns.yaml):

apiVersion: v1

kind: Namespace

metadata:

creationTimestamp: null

name: security

**Create the ConfigMap for the service (config-uaa.yaml):**

apiVersion: v1

kind: ConfigMap

metadata:

name: uaa-config

namespace: security

data:

uaa.yml: |

uaa:

url: http://gav.wsl2

issuer:

uri: http://gav.wsl2/uaa

encryption:

encryption\_keys:

- label: uaa-encryption-key-1

passphrase: ((uaa\_encryption\_key\_1))

active\_key\_label: uaa-encryption-key-1

oauth:

user:

authorities:

- openid

- scim.me

- password.write

- scim.userids

- uaa.user

- approvals.me

- oauth.approvals

clients:

proxy\_gateway:

name: proxy\_gateway

secret: thisissecret

authorized-grant-types: authorization\_code

scope: openid,profile,email,resource.read

authorities: uaa.resource

#redirect-uri: http://gav.wsl2/login/oauth2/code/proxy\_gateway

redirect-uri: https://gav.wsl2/oauth2/callback

scim:

groups:

email: Access your email address

resource.read: Allow access with 'resource.read'

users:

- gavin|!234Qwer|gavin.pollitt@capgemini.co.uk|Gavin|Pollitt|uaa.user,profile,email,resource.read

- [gavlad|!234Qwer|gavin.pollitt@capgemini.co.uk|Gavladdio|Pollitt|uaa.user,profile,email,resource.read](mailto:gavlad|!234Qwer|gavin.pollitt@capgemini.co.uk|Gavladdio|Pollitt|uaa.user,profile,email,resource.read)

…various key set-up

**And, now, create the actual deployment of the UAA service that will mount the above config (uaa-dep.yaml):**

apiVersion: apps/v1

kind: Deployment

metadata:

creationTimestamp: null

labels:

app: uaa-dep

name: uaa-dep

namespace: security

spec:

replicas: 1

selector:

matchLabels:

app: uaa-dep

strategy: {}

template:

metadata:

creationTimestamp: null

labels:

app: uaa-dep

spec:

hostAliases:

- ip: 172.18.120.197

hostnames:

- "gav.wsl2"

volumes:

- name: conf-vol

configMap:

name: uaa-config

containers:

- image: gavinpollitt/uaa:1.0

name: uaa

resources: {}

volumeMounts:

- name: conf-vol

mountPath: /usr/local/tomcat/temp/uaa

ports:

- containerPort: 8090

status: {}

**Expose with a service (uaa-svc.yaml):**

apiVersion: v1

kind: Service

metadata:

creationTimestamp: null

labels:

app: uaa-dep

name: uaa-dep

namespace: security

spec:

ports:

- port: 8090

protocol: TCP

targetPort: 8090

name: gui

- port: 8080

protocol: TCP

targetPort: 8080

name: access

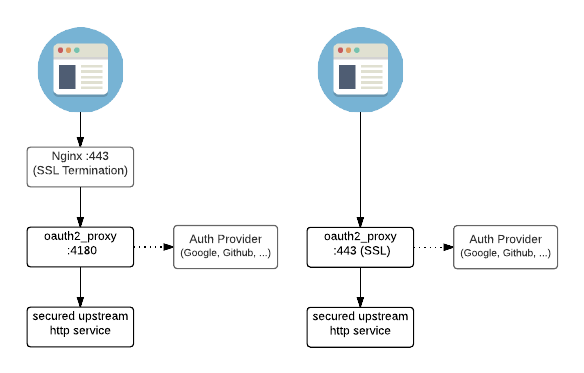
selector:

app: uaa-dep

type: NodePort

status:

**To enable an Ingress solution with our UAA service, we need a proxy to choreograph the OAuth flow. We use the ‘oauth2-proxy’ container to allow the following example to be implemented:**



**Here’s our deployment for the oauth2-proxy service (proxy.yaml):**

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

k8s-app: oauth2-proxy

name: oauth2-proxy

namespace: kube-system

spec:

replicas: 1

selector:

matchLabels:

k8s-app: oauth2-proxy

template:

metadata:

labels:

k8s-app: oauth2-proxy

spec:

hostAliases:

- ip: 172.18.120.197

hostnames:

- "gav.wsl2"

containers:

- args:

- --provider=oidc

- --oidc-issuer-url=http://gav.wsl2/uaa/oauth/token

- --email-domain=\*

- --provider-display-name="UAA Provider"

- --redirect-url=https://gav.wsl2/oauth2/callback

- --client-id=proxy\_gateway

- --client-secret=thisissecret

- --cookie-secure=false

- --http-address=0.0.0.0:4180

env:

- name: OAUTH2\_PROXY\_CLIENT\_ID

value: proxy\_gateway

- name: OAUTH2\_PROXY\_CLIENT\_SECRET

value: thisissecret

# docker run -ti --rm python:3-alpine python -c 'import secrets,base64; print(base64.b64encode(base64.b64encode(secrets.token\_bytes(16))));'

- name: OAUTH2\_PROXY\_COOKIE\_SECRET

value: OHNLcEQ5Y3ZiZlJxMjkrb2ZkN2REQT09

image: quay.io/oauth2-proxy/oauth2-proxy:latest

imagePullPolicy: Always

name: oauth2-proxy

ports:

- containerPort: 4180

protocol: TCP

**And, expose the service itself to the cluster only (proxy.yaml):**

apiVersion: v1

kind: Service

metadata:

labels:

k8s-app: oauth2-proxy

name: oauth2-proxy

namespace: kube-system

spec:

ports:

- name: http

port: 4180

protocol: TCP

targetPort: 4180

selector:

k8s-app: oauth2-proxy

**Now, we need to adjust our Ingress set-up accordingly (ingress-override.yaml):**

**Firstly, delete the current client-ingress from the client namespace:**

kubectl apply -f ingress.yaml (in the ingress directory)

**and relocate the ‘gavq’ secret to the kube-system namespace:**

kubectl delete secret gavq -n client

**then run:** keys-override.sh

**Now apply the more detailed Ingress requirements:**

**Firstly, re-apply our client-ingress, but now to direct to our oauth2-proxy for any path:**

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

annotations:

kubernetes.io/ingress.class: "nginx"

name: client-ingress

namespace: kube-system

spec:

tls:

- hosts:

- gav.wsl2

secretName: gavq

rules:

- host: gav.wsl2

http:

paths:

- backend:

serviceName: oauth2-proxy

servicePort: 4180

path: /oauth2

**And, we need ingress for our UAA service for the redirect that will be sent to the user browser:**

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

annotations:

kubernetes.io/ingress.class: "nginx"

nginx.ingress.kubernetes.io/ssl-redirect: "false"

name: uaa-ingress

namespace: security

spec:

rules:

- host: gav.wsl2

http:

paths:

- path: /uaa

backend:

serviceName: uaa-dep

servicePort: 8090

path: /uaa

**Finally, ensure that out /queue is secured by the proxy.**

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: queue-ingress

namespace: client

annotations:

kubernetes.io/ingress.class: "nginx"

nginx.ingress.kubernetes.io/auth-url: "https://$host/oauth2/auth"

nginx.ingress.kubernetes.io/auth-signin: "https://$host/oauth2/start?rd=$escaped\_request\_uri"

labels:

app: queue-ingress

spec:

rules:

- host: gav.wsl2

http:

paths:

- path: /queue

backend:

serviceName: qclient-dep

servicePort: 8080

**Now, add the following into a browser (probably best private window):** [https://gav.wsl2/queue/getCompany?userName="gav](https://gav.wsl2/queue/getCompany?userName=%22gav)" **accepting any security warnings.**

**Log in:** gavin/!234Qwer

**And you should then be re-directed to received the queue contents (if any).**