KUBERNETES FOR DEVOPS

ANISH NATH

Kubernetes for DevOps

Anish Nath

This book is for sale at http://leanpub.com/kube

This version was published on 2019-11-20

Leanpub

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

I would like to dedicate this book to my 8-year old son Arjun Nath , Grandfather Sri Rajeshwar Prasad; wife Manisha Prasad; mother Indu Sinha; and all my family members (my father Anil Kumar Sinha; chote papa Sunil Kumar Sinha; choti mummy: Poonam Sinha; and friends). Without them, this would not have been possible.

I would also thanks to Cisco where I work, I got most of the learning from there, and not least the opensource community.

Contents

Setup nginx-ingress and Route application 1
Setting Up Role-Based Access Control 13

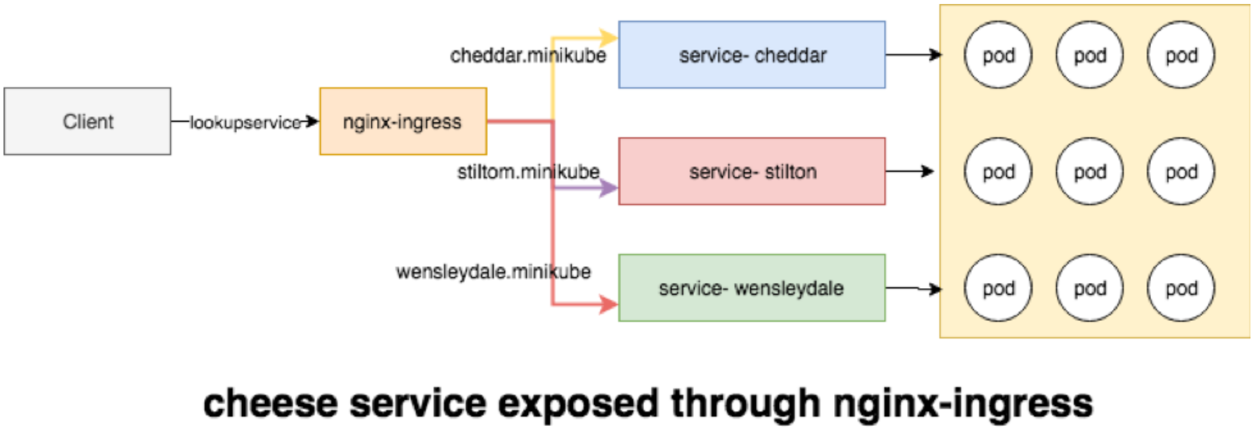
Setup nginx-ingress and Route application

Kubernetes is the great way to manage docker service in the orchestration way, the service which are created need to be exposed to external clients, which can be done in many ways, This tutorial explains how to use nginx-ingress as an Ingress controller for a Kubernetes cluster and covers topic like

- Setting up nginx-ingress
- Define Name based Routing
- Define Path Based routing

About nginx-ingress

An Ingress Controller is a daemon, deployed as a Kubernetes Pod, that watches the apiserver's `/ingresses` endpoint for updates to the Ingress resource[1]. Its job is to satisfy requests for Ingresses.



The LAB application

Install nginx-ingress using Helm

The below command will setup the **nginx-ingress** in the **kube-system** namespace, and setup necessary RBAC in k8 cluster to operated ingress correctly.

```
root@kube-master:#  helm install stable/nginx-ingress --name nginx-ingress --set con\
troller.stats.enabled=true --namespace kube-system
```

This command produces a **lot of output**, so let's take it one step at a time. First, we get information about the release that's been deployed

[1] https://kubernetes.io/docs/concepts/services-networking/ingress/

```
NAME: nginx-ingress
LAST DEPLOYED: Thu Jan 24 14:00:16 2019
NAMESPACE: kube-system
STATUS: DEPLOYED
```

Next we get the resources that were actually deployed by the **stable/nginx-ingress** chart

```
RESOURCES:
==> v1beta1/ClusterRoleBinding
NAME           AGE
nginx-ingress  4s

==> v1beta1/RoleBinding
NAME           AGE
nginx-ingress  4s

==> v1/Service
NAME                            TYPE          CLUSTER-IP    EXTERNAL-IP  PORT(S)    \
                 AGE
nginx-ingress-controller        LoadBalancer  10.98.81.209  <pending>    80:32437/TC\
P,443:30692/TCP  4s
nginx-ingress-controller-stats  ClusterIP     10.96.0.80    <none>       18080/TCP  \
                 4s
nginx-ingress-default-backend   ClusterIP     10.106.7.213  <none>       80/TCP     \
                 4s

==> v1beta1/ClusterRole
NAME           AGE
nginx-ingress  4s

==> v1/ServiceAccount
NAME           SECRETS  AGE
nginx-ingress  1        4s

==> v1beta1/Role
NAME           AGE
nginx-ingress  4s

==> v1beta1/Deployment
NAME                           DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
nginx-ingress-controller       1        1        1           0          4s
nginx-ingress-default-backend  1        1        1           0          4s
```

```
==> v1/Pod(related)
NAME                                          READY  STATUS             RESTARTS  \
AGE
nginx-ingress-controller-ff7cb987-lj4j5       0/1    Pending            0         \
3s
nginx-ingress-default-backend-544cfb69fc-xtl2p  0/1    ContainerCreating  0         \
4s

==> v1/ConfigMap
NAME                      DATA  AGE
nginx-ingress-controller  1     4s
```

The chart also enables the developer to add notes:

```
NOTES:
The nginx-ingress controller has been installed.
It may take a few minutes for the LoadBalancer IP to be available.
You can watch the status by running 'kubectl --namespace kube-system get services -o\
 wide -w nginx-ingress-controller'

An example Ingress that makes use of the controller:

  apiVersion: extensions/v1beta1
  kind: Ingress
  metadata:
    annotations:
      kubernetes.io/ingress.class: nginx
    name: example
    namespace: foo
  spec:
    rules:
      - host: www.example.com
        http:
          paths:
            - backend:
                serviceName: exampleService
                servicePort: 80
              path: /
    # This section is only required if TLS is to be enabled for the Ingress
    tls:
        - hosts:
            - www.example.com
          secretName: example-tls
```

```
If TLS is enabled for the Ingress, a Secret containing the certificate and key must \
also be provided:

  apiVersion: v1
  kind: Secret
  metadata:
    name: example-tls
    namespace: foo
  data:
    tls.crt: <base64 encoded cert>
    tls.key: <base64 encoded key>
  type: kubernetes.io/tls
```

Well if everything goes well then, Check the **nginx-ingress** pods are **running** in the kube-system namespace

```
root@kube-master:# kubectl --namespace kube-system get services -o wide -w nginx-ing\
ress-controller
NAME                       TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)      \
                AGE       SELECTOR
nginx-ingress-controller   LoadBalancer   10.98.81.209   172.16.2.13   80:32437/TCP,\
443:30692/TCP   3m        app=nginx-ingress,component=controller,release=nginx-ingre\
ss
```

Once '**EXTERNAL-IP**' is no longer '<pending>': your **nginx-ingress** is ready for use

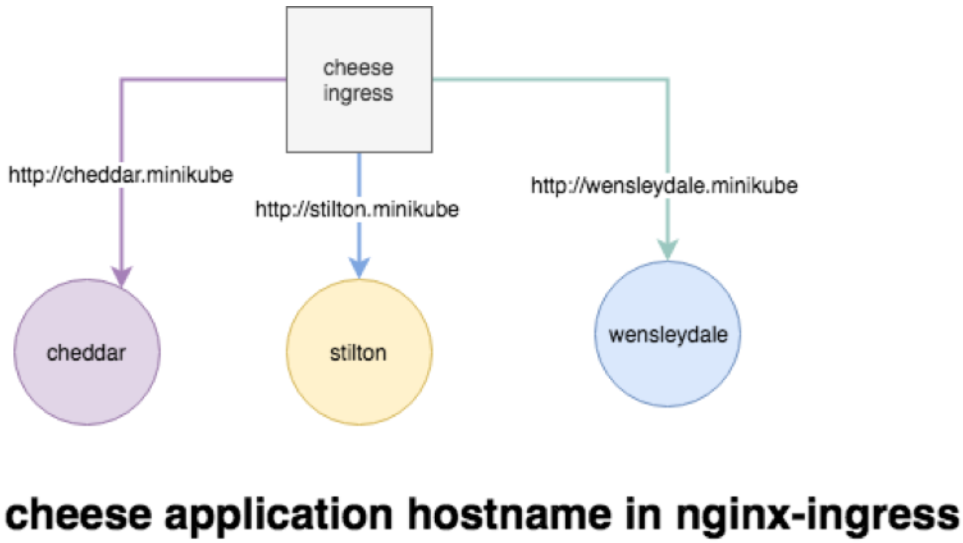
Up to this point we have successfully installed and configured **nginx-ingress** , now let's define the routing (**name based and path based**) of your application.

The Demo Application

We are going here to setup three sample nginx cheese web application, the docker images are located here[2] .

1. Docker image: errm/cheese:wensleydale
2. Docker image: errm/cheese:cheddar
3. Docker image: errm/cheese:stilton

[2] https://hub.docker.com/r/errm/cheese/tags



Demo Application

Name based routing

The Name-Based Routing performs routing by name and support routing HTTP traffic to multiple host names at the same IP address but different domain names , let’s start by launching the pods for the cheese websites.

Deployment of Cheese Web Application

The YAML file for the cheese application

```
---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: stilton
  labels:
    app: cheese
    cheese: stilton
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cheese
      task: stilton
  template:
    metadata:
      labels:
```

```
      app: cheese
      task: stilton
      version: v0.0.1
  spec:
    containers:
    - name: cheese
      image: errm/cheese:stilton
      resources:
        requests:
          cpu: 100m
          memory: 50Mi
        limits:
          cpu: 100m
          memory: 50Mi
      ports:
      - containerPort: 80
```

To provide some explanations about the file content:

- We define a deployment (`kind: Deployment`)
- The name of the object is “stilton” (`name: stilton`)
- We want one replica (`replicas: 2`)
- It will deploy pods that have the label app:cheese (`selector: matchLabels: app:cheese`)
- Then we define the pods (`template: ...`)
- The Pods will have the cheese label (`metadata:labels:app:cheese`)
- The Pods will host a container using the image tag errm/cheese:stilton (`image: errm/cheese:stilton`)
- The same deployment is repeated for cheddar and wensleydale

Now provision these nginx cheese application

```
root@kube-master:# kubectl apply -f https://raw.githubusercontent.com/anishnath/kube\
rnetes/master/cheese-deployments.yaml
deployment.extensions/stilton created
deployment.extensions/cheddar created
deployment.extensions/wensleydale created
```

Make sure all the cheese deployment pods are up and running.

```
root@kube-master:# kubectl get pods
NAME READY STATUS  RESTARTS AGE
cheddar-6c895c7cc7-2qztp 1/1 Running 0  7m
cheddar-6c895c7cc7-mzq9v 1/1 Running 0  7m
stilton-7989d7c86f-62wrt 1/1 Running 0  7m
stilton-7989d7c86f-fjttz 1/1 Running 0  7m
wensleydale-58784fc6f7-f8szd 1/1 Running 0  7m
wensleydale-58784fc6f7-prb8z 1/1 Running 0  7m
```

Now Service the Cheese Web Application

```
root@kube-master:# kubectl apply -f https://raw.githubusercontent.com/anishnath/kube\
rnetes/master/cheese-services.yaml
service/stilton created
service/cheddar created
service/wensleydale created
```

Check all the necessary service is created in k8 cluster.

```
root@kube-master:# kubectl get svc
NAME  TYPE  CLUSTER-IP EXTERNAL-IP PORT(S) AGE
cheddar ClusterIP 10.108.200.238 <none>  80/TCP  30s
kubernetes  ClusterIP 10.96.0.1  <none>  443/TCP 1h
stilton ClusterIP 10.102.20.8  <none>  80/TCP  30s
wensleydale ClusterIP 10.109.58.21 <none>  80/TCP  30s
```

At this point, we have deployment and Service ready in the K8 cluster, and we're about to define the ingress rules so that the world can eat the required service.

```
root@kube-master:# echo "
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: cheese
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - host: stilton.minikube
    http:
      paths:
      - path: /
    backend:
      serviceName: stilton
      servicePort: http
 - host: cheddar.minikube
   http:
     paths:
     - path: /
       backend:
         serviceName: cheddar
         servicePort: http
 - host: wensleydale.minikube
   http:
     paths:
     - path: /
       backend:
         serviceName: wensleydale
         servicePort: http
" | kubectl apply -f -
```

The command output

```
ingress.extensions/cheese created
```

To provide some explanations about the file content:

- We define a Ingress (`kind: Ingress`) and add ingress class in the annotation
- The name of the object is “cheese” (`name: cheese`)
- Then we define the rules (`rules: ...`)
- For each service there is hostname defined for example the hostname **stilton.minikube** is mapped to **stilton** service.
- The rules are repeated for each service.

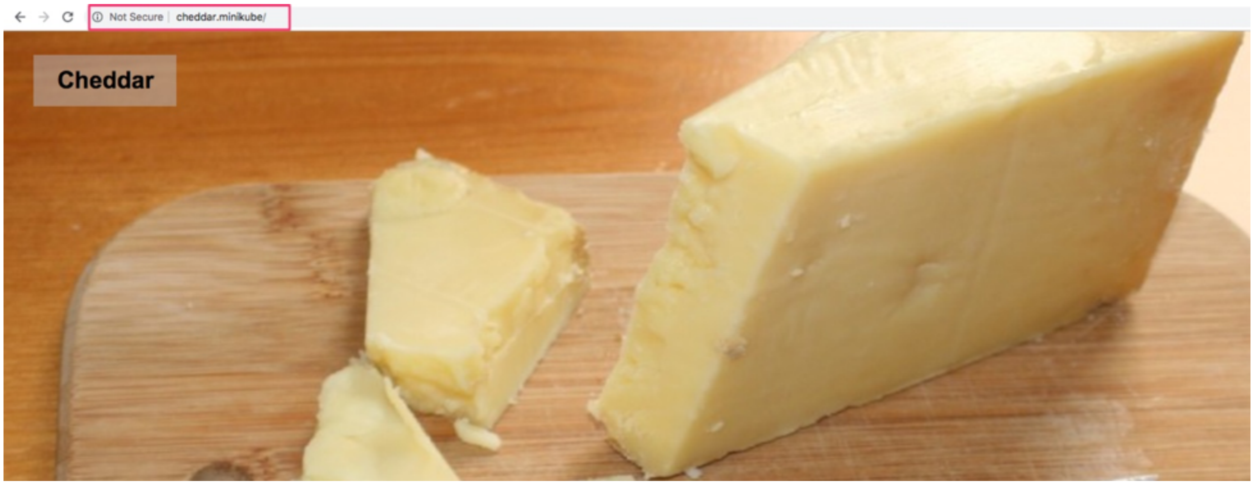
Verify the Ingress, all the hosts can be accessed with the ingress port 80

```
root@kube-master:# kubectl get ingress
NAME   HOSTS   ADDRESS PORTS AGE
cheese  stilton.minikube,cheddar.minikube,wensleydale.minikube 80  31s
```

Testing

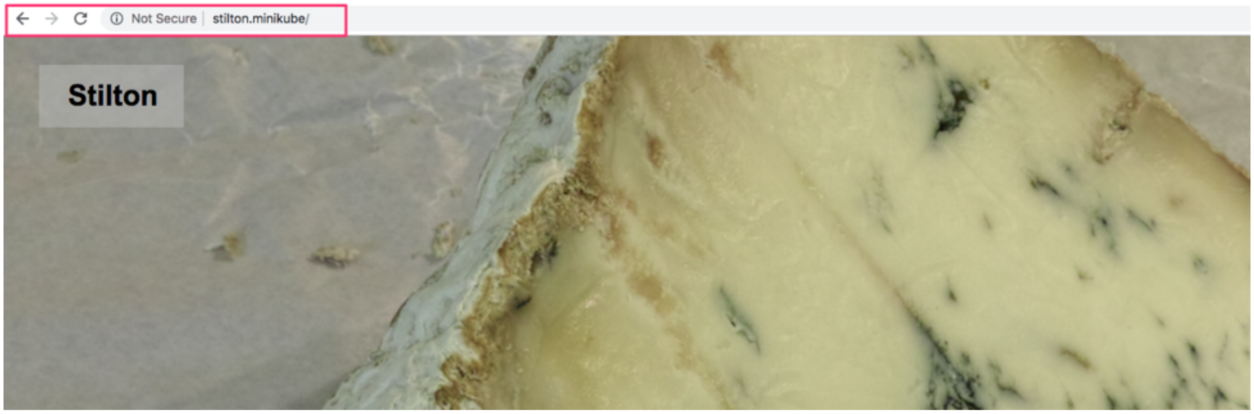
Open the web browser and start eating your favorite cheese

- http://cheddar.minikube/[3]



cheddar

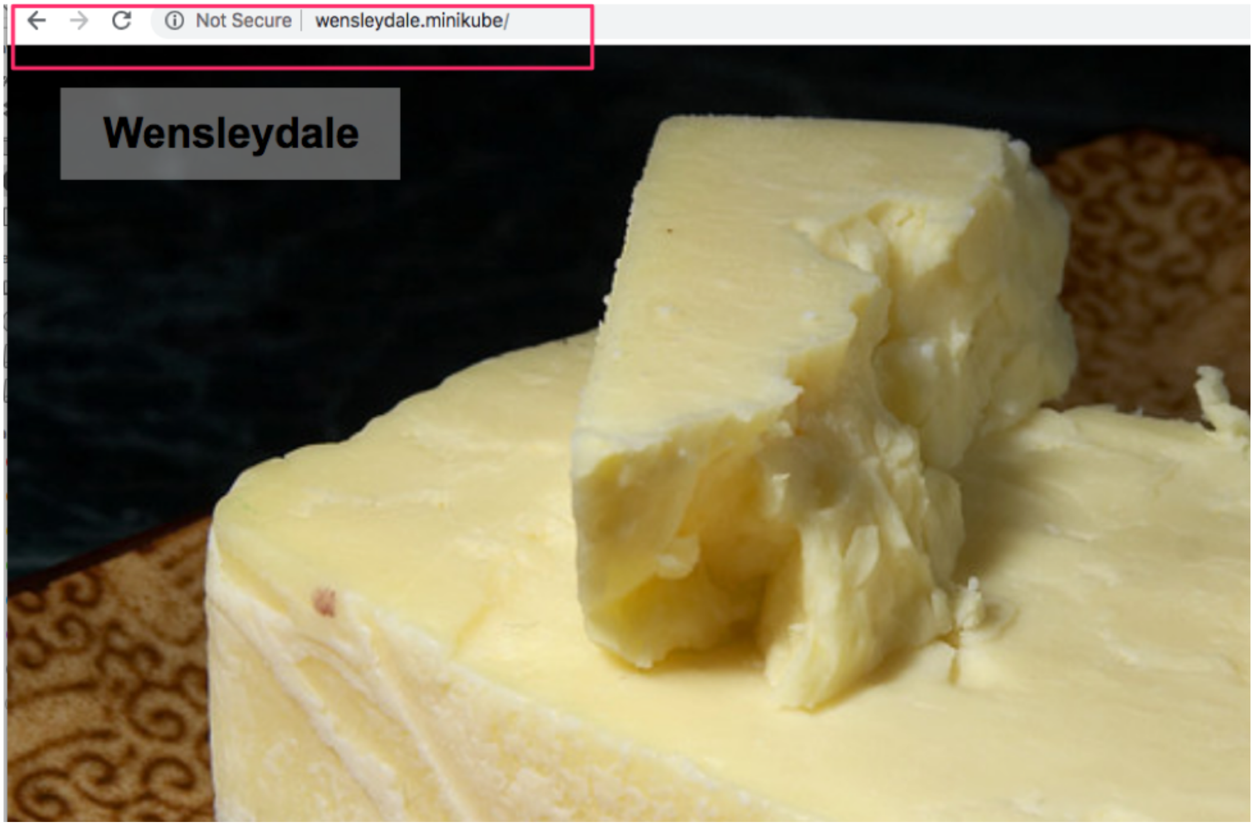
- http://stilton.minikube/[4]



stilton

- http://wensleydale.minikube/[5]

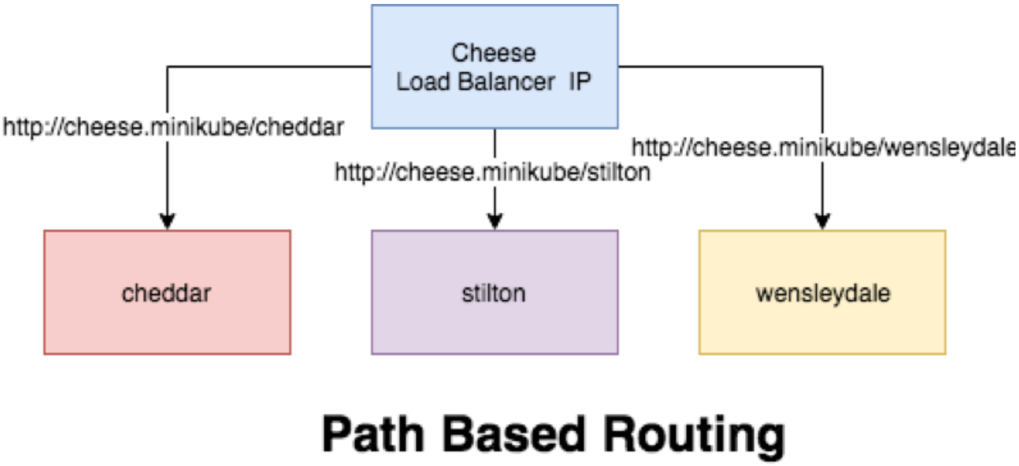
[3] http://cheddar.minikube/
[4] http://stilton.minikube/
[5] http://wensleydale.minikube/



wensleydale

PATH based routing

Path based routing differ from Name based routing in a sense, we don’t have multiple domains names, all the URI is distinguished and routed from the PATH prefix under a single domain, for example the above cheese application can be access through the single URI.


Cheese
Load Balancer IP
http://cheese.minikube/cheddar
http://cheese.minikube/stilton
http://cheese.minikube/wensleydale
cheddar
stilton
wensleydale
Path Based Routing

PATH Bases routing

Let’s create the PATH base routing for the cheese application by defining the ingress

```
root@kube-master:# echo "
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: cheeses
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: cheeses.minikube
    http:
      paths:
      - path: /stilton
        backend:
          serviceName: stilton
          servicePort: http
      - path: /cheddar
        backend:
          serviceName: cheddar
          servicePort: http
      - path: /wensleydale
        backend:
          serviceName: wensleydale
          servicePort: http
" | kubectl apply -f -
```

The Command output

```
ingress.extensions/cheese created
```

You should now be able to visit the websites in your browser.

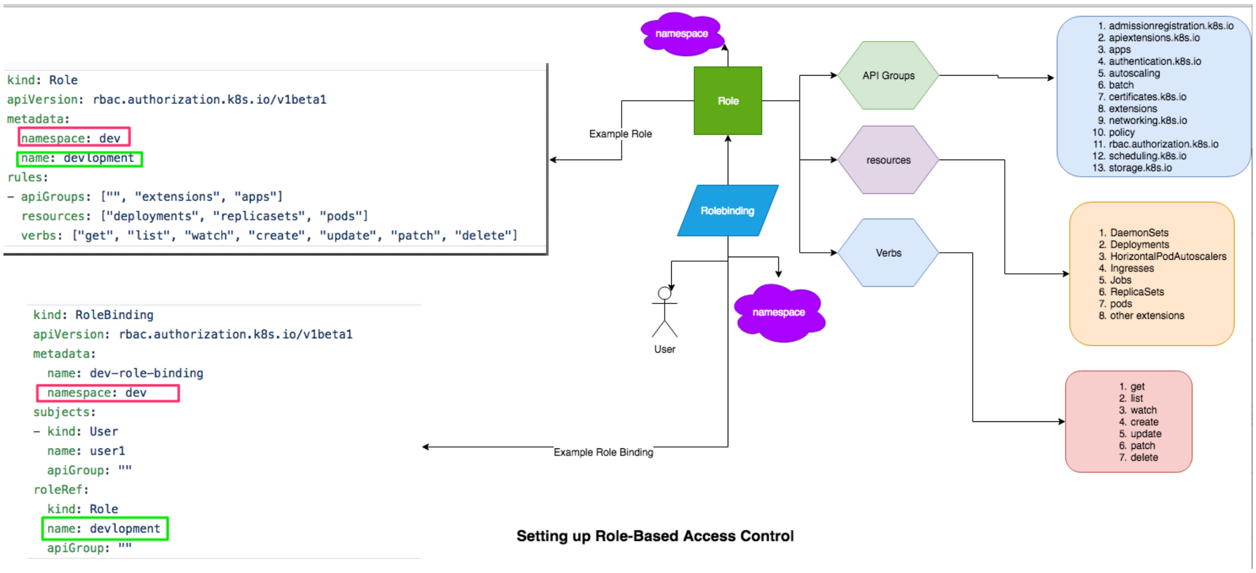
- http://cheeses.minikube/stilton[6]
- http://cheeses.minikube/cheddar[7]
- http://cheeses.minikube/wensleydale[8]

Final Note

- The above example doesn't use any SSL configuration
- It is advisable to install the nginx-ingress in the kube-system namespace
- Always measure your resource needs, and adjust requests and limits accordingly.

[6] http://cheeses.minikube/stilton/
[7] http://cheeses.minikube/cheddar/
[8] http://cheeses.minikube/wensleydale/

Setting Up Role-Based Access Control



```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: dev
  name: devlopment
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dev-role-binding
  namespace: dev
subjects:
- kind: User
  name: user1
  apiGroup: ""
roleRef:
  kind: Role
  name: devlopment
  apiGroup: ""
```

Setting up Role-Based Access Control

RBAC

You define your RBAC permissions by creating objects from the `rbac.authorization.k8s.io` API group in your cluster. You can create the objects using the `kubectl` command-line interface, or programmatically.

You'll need to create two kinds of objects:

1. A `Role` or `ClusterRole` object that defines what resource types and operations are allowed for a set of users.
2. A `RoleBinding` or `ClusterRoleBinding` that associates the `Role` (or `ClusterRole`) with one or more specific users.

RBAC permissions are purely additive there are no "deny" rules. When structuring your RBAC permissions, you should think in terms of "granting" users access to cluster resources.

The LAB

In this LAB exercise, we are going to run the below use case

- Create namespaces **dev** and **stag**
- Create two user names **user1** and **user2**
- **user1** belongs to **dev** namespace
- **user2** belongs to **stage** namespace
- user1 and user2 defined with **Role** and **RoleBinding**
- **user1** created **busybox** pod in dev namespace

- **user2** created **busybox** pod in stage namespace
- **user1** tried to access **busybox** pod in **stage** namespace **Access Denied** (Valid Use case)
- **user2** tried to access **busybox** pod in **dev** namespace **Access Denied** (Valid Use case)
- **user1** can query pods in **dev** namespace (Valid Use case)
- **user2** can query pod in **stage** namespace (Valid Use case)

1. Creating dev and stage namespace

To learn more about namespaces go here[9]

```
root@kube-master:# kubectl create namespace dev
namespace/dev created
root@kube-master:# kubectl create namespace stage
namespace/stag created
```

2. Creating user1

- To create **user1** generate RSA keys for user1 create CSR and get it signed with Kubernetes rootCA and rootCA private key.

Generate RSA keys for user1

```
root@kube-master:# openssl genrsa -out user1.key 2048
Generating RSA private key, 2048 bit long modulus
..................................................................................+++
.................+++
e is 65537 (0x10001)
```

Generate the CSR for user1

```
root@kube-master:# openssl req -new -key user1.key -out user1.csr -subj "/CN=user1/O\
=8gwifi.org"
```

Sign the CSR and create the user1 x.509 certificate, sign CSR with the Kubernetes rootCA and rootCA key which usually present in the `/etc/kubernetes/pki/` location.

[9] kube-namespaces.jsp

```
root@kube-master:# openssl x509 -req -in user1.csr -CA /etc/kubernetes/pki/ca.crt -C\
Akey /etc/kubernetes/pki/ca.key -CAcreateserial -out user1.crt -days 365
Signature ok
subject=/CN=user1/O=8gwifi.org
Getting CA Private Key
```

update the kubernetes config and define `set-credentials` and `set-context` for **user1**

```
root@kube-master:# kubectl config set-credentials user1 --client-certificate=user1.c\
rt --client-key=user1.key
User "user1" set.
root@kube-master:# kubectl config set-context dev --cluster=kubernetes --namespace=d\
ev --user=user1
Context "dev" modified.
```

3. Creating user2

Repeat the same process for creating **user2** in Kubernetes cluster

```
openssl genrsa -out user2.key 2048
openssl req -new -key user2.key -out user2.csr -subj "/CN=user2/O=8gwifi.org"
openssl x509 -req -in user2.csr -CA /etc/kubernetes/pki/ca.crt -CAkey /etc/kubernete\
s/pki/ca.key -CAcreateserial -out user2.crt -days 365
```

update the kubernetes config and define `set-credentials` and `set-context` for **user2**

```
kubectl config set-credentials user2 --client-certificate=user2.crt --client-key=use\
r2.key
kubectl config set-context stage --cluster=kubernetes --namespace=stage --user=user2
```

4. Create Role and Rolebinding for user1

Creating role Create a role in **dev** namespace, In this *yaml* file we are creating the rule that allows a user to execute operations like **deployments**,**replicasets**,**pods**

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: dev
  name: devlopment
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

Apply this role in kubernetes cluster

```
root@kube-master:# kubectl create -f dev-role.yaml
role.rbac.authorization.k8s.io/devlopment created
```

Bind this role to user1

Binding the **user1** to the `Role:development`

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dev-role-binding
  namespace: dev
subjects:
- kind: User
  name: user1
  apiGroup: ""
roleRef:
  kind: Role
  name: devlopment
  apiGroup: ""
```

Apply this role binding in Kubernetes cluster

```
root@kube-master:# kubectl create -f  dev-role-binding.yaml
rolebinding.rbac.authorization.k8s.io/dev-role-binding created
```

5. Create Role and Rolebinding for user2

Repeat the same process for **user2**, in the **stage** namespace, creating **role** in **stage** namespace

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: stage
  name: staging
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

Apply the role in k8 cluster

```
$ kubectl create -f stage-role.yaml
```

Create role binding for user2

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: stage-role-binding
  namespace: stage
subjects:
- kind: User
  name: user2
  apiGroup: ""
roleRef:
  kind: Role
  name: staging
  apiGroup: ""
```

```
root@kube-master:# kubectl create -f stage-role-binding.yaml
```

6. Verify Roles and RoleBindings

Verify the namespace points to correct **role,rolebindings and users**

```
root@kube-master:# kubectl get roles -n dev
root@kube-master:# kubectl get roles -n stage
root@kube-master:# kubectl get rolebinding -n stage
root@kube-master:# kubectl get rolebinding -n dev
```

```
root@kube-master:# kubectl describe rolebinding dev-role-binding -n dev
Name:         dev-role-binding
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  Role
  Name:  devlopment
Subjects:
  Kind  Name   Namespace
  ----  ----   ---------
  User  user1
root@kube-master:# kubectl describe rolebinding stage-role-binding -n stage
Name:         stage-role-binding
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  Role
  Name:  staging
Subjects:
  Kind  Name   Namespace
  ----  ----   ---------
  User  user2
```

8. Launch busybox pods in the respective namespace

In practice you can launch any deployment here, the busybox is chosen for testing purpose only

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
```

Creating busybox pods in **stage** and **dev** namespaces

```
root@kube-master:# kubectl create -f busybox.yaml -n stage
pod/busybox created
root@kube-master:# kubectl create -f busybox.yaml -n dev
pod/busybox created
```

9. Testing RBAC

While creating **user1** and **user2** the config context are set, verify it's working as desired, this is also used for RBAC troubleshooting.

```
root@kube-master:# kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
          dev                           kubernetes   user1              dev
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
          stage                         kubernetes   user2              stage
```

- Valid Use case by using dev and stage context both user1 and user2 will see their respective pods.

```
root@kube-master:# kubectl --context=dev get pods
NAME      READY     STATUS    RESTARTS   AGE
busybox   1/1       Running   0          4m
root@kube-master:# kubectl --context=stage get pods
NAME      READY     STATUS    RESTARTS   AGE
busybox   1/1       Running   0          4m
```

- Valid use case, **user2** will be forbidden to check on **dev** context

```
root@kube-master:# kubectl --context=dev get pods --user=user2
No resources found.
Error from server (Forbidden): pods is forbidden: User "user2" cannot list pods in t\
he namespace "dev"
```

- Valid use case **user1** will be forbidden to access **stage** context

```
root@kube-master:# kubectl --context=stage get pods --user=user1
No resources found.
Error from server (Forbidden): pods is forbidden: User "user1" cannot list pods in t\
he namespace "stage"
```