Kubernetes Ingress and Ingress controller

What is Ingress?

Definition of Ingress:

An Ingress is a Kubernetes resource that manages external access to services within a Kubernetes cluster.

It routes HTTP/S traffic from outside the cluster to the appropriate services inside the cluster based on rules defined within the Ingress resource.

Components of Ingress:

Host: The domain name or IP address that identifies the resource.

Path: A specific URL path for routing the traffic (e.g., /api).

Backend Services: The services that traffic should be directed to based on the defined rules.

How Ingress Works in Kubernetes?

Ingress Resource:

The Ingress resource specifies the routing rules, such as which host and path should be directed to which service.

Ingress resources are created using YAML configuration files.

Traffic Routing:

When an external client makes an HTTP request, the Ingress Controller reads the rules defined in the Ingress resource and forwards the traffic to the corresponding service based on the defined path and host.

TLS/SSL Termination:

Ingress can also manage SSL/TLS termination by providing the ability to define certificates and terminate secure connections at the Ingress Controller level.

Ingress Controllers

Definition of Ingress Controller:

An Ingress Controller is a Kubernetes controller that manages the Ingress resources and handles the traffic routing.

It is a load balancer that listens for changes in the Ingress resources and adjusts the configuration accordingly.

Role in Traffic Management:

The Ingress Controller is responsible for managing traffic routing based on the rules defined in the Ingress resources.

It can also provide additional features like load balancing, SSL termination, and path-based routing.

Popular Ingress Controllers

NGINX Ingress Controller:

A widely-used Ingress controller based on NGINX, it provides features like load balancing, SSL termination, and path-based routing.

Supports custom configurations and advanced features such as rate limiting and authentication.

Traefik:

A modern, dynamic, and highly configurable Ingress controller.

Automatically discovers services and supports websocket and GRPC protocols.

HAProxy:

Another widely-used option for Ingress control, focusing on high performance and robustness.

Supports advanced features like load balancing, sticky sessions, and HTTP/2.

Envoy:

Envoy is an open-source proxy that can act as an Ingress Controller, often used in service meshes.

Provides features like circuit breaking, rate limiting, and load balancing.

Prerequisites

You should already have the following installed and configured:

- ✓ kops installed and configured
- ✓ kubectl installed and configured
- ✓ AWS CLI (aws) installed and configured with IAM permissions
- ✓ An S3 bucket created and used for Kops state store
- ✓ A Kubernetes cluster created via Kops (1 master + 1 node)

Step 1: Deploy a sample app with Deployment + Service + Ingress (NGINX)

Sample Deployment YAML (httpd)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpd-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      containers:
      - name: httpd
        image: httpd:2.4
        ports:
        - containerPort: 80
```

Service YAML to expose the Deployment

```
apiVersion: v1
kind: Service
metadata:
  name: httpd-service
spec:
  selector:
    app: httpd
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: ClusterIP
```

Ingress YAML to expose service externally via NGINX ingress controller

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: httpd-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx   # use the ingress controller class name
  rules:
  - host: example.yourdomain.com   # replace with your domain or use /etc/hosts for testing
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: httpd-service
            port:
              number: 80
```

Step 2: How to deploy and use

1. **Apply the Deployment:**

   ```
   kubectl apply -f deployment.yaml
   ```

2. **Apply the Service:**

   ```
   kubectl apply -f service.yaml
   ```

3. **Apply the Ingress:**

   ```
   kubectl apply -f ingress.yaml
   ```

Step 3: Verify resources

- Check pods are running:

```
kubectl get pods -l app=httpd
```

- Check service is created:

```
kubectl get svc httpd-service
```

- Check ingress is created:

```
kubectl get ingress httpd-ingress
```

- Check the external IP or hostname of the NGINX ingress controller load balancer:

```
kubectl get svc -n ingress-nginx
```

Note the **EXTERNAL-IP** of the ingress-nginx-controller service. This is where you point your domain or test.

Step 4: Access your app

- If you have a domain like example.yourdomain.com, set DNS A record to the external IP from above.
- Or add to your /etc/hosts for testing:

<EXTERNAL-IP> example.yourdomain.com

- Then access:

http://example.yourdomain.com

You should see the default httpd welcome page served by your Kubernetes cluster.

Step 5: What happens?

- Your browser calls example.yourdomain.com at the external IP.
- NGINX ingress controller receives the request.
- Based on the Ingress rules, it routes traffic to the httpd-service on port 80.
- Service forwards request to one of the httpd pods.
- Pod responds and the response reaches your browser.

To delete all resources after testing

kubectl delete -f ingress.yaml
kubectl delete -f service.yaml
kubectl delete -f deployment.yaml

1. Host-Based Routing Ingress Example

Use case:

Route requests to different services based on the domain name.

Sample setup:

Assuming you have two services:

- app1-service (running app1)
- app2-service (running app2)

Ingress YAML:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: host-based-ingress
spec:
  ingressClassName: nginx
  rules:
  - host: app1.example.com
```

```
  http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: app1-service
          port:
            number: 80
- host: app2.example.com
  http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: app2-service
          port:
            number: 80
```

How to test:

- Add the domains in /etc/hosts pointing to your ingress controller external IP:

<EXTERNAL-IP> app1.example.com
<EXTERNAL-IP> app2.example.com

- Access http://app1.example.com → routes to app1-service
- Access http://app2.example.com → routes to app2-service

2. Path-Based Routing Ingress Example

Use case:

Route requests to different services based on the URL path.

Ingress YAML:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: path-based-ingress
spec:
  ingressClassName: nginx
  rules:
  - host: example.com
    http:
      paths:
      - path: /app1
```

```
      pathType: Prefix
      backend:
       service:
         name: app1-service
         port:
           number: 80
    - path: /app2
      pathType: Prefix
      backend:
       service:
         name: app2-service
         port:
           number: 80
```

How to test:

- Add in /etc/hosts:

arduino
<EXTERNAL-IP> example.com

- Access:
  - http://example.com/app1 → routes to app1-service
  - http://example.com/app2 → routes to app2-service

3. TLS / HTTPS with Ingress Example

Step 1: Create TLS Secret

Assuming you have your cert files tls.crt and tls.key:

```
kubectl create secret tls example-tls --cert=tls.crt --key=tls.key --namespace=default
```

Step 2: Ingress YAML with TLS

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tls-ingress
spec:
  ingressClassName: nginx
  tls:
  - hosts:
    - secure.example.com
    secretName: example-tls
  rules:
```

```
  - host: secure.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: secure-service
            port:
              number: 443
```

Step 3: Test

- Update /etc/hosts:

```
<EXTERNAL-IP> secure.example.com
```

- Access:
  - https://secure.example.com (browser should show secured connection)

4. Useful NGINX Ingress Annotations (Examples)

Annotation	Purpose
nginx.ingress.kubernetes.io/rewrite-target	Rewrite the URL path (e.g. /foo to /)
nginx.ingress.kubernetes.io/ssl-redirect	Enable/disable HTTP → HTTPS redirect (true/false)
nginx.ingress.kubernetes.io/proxy-body-size	Max request body size (e.g., 8m)
nginx.ingress.kubernetes.io/whitelist-source-range	Allow traffic only from certain IPs (e.g., 1.2.3.4/32)

Example snippet:

```
metadata:
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
```

How to Debug Kubernetes Ingress Problems

1. Check the Ingress Controller Pod Status

Make sure the ingress controller pods are running and healthy.

kubectl get pods -n ingress-nginx

Look for pods in Running status, no restarts or crashes.

2. Check Ingress Resource Details

Check if the Ingress resource exists and is correctly configured:

kubectl describe ingress <ingress-name>

Look for:

- Events section (errors or warnings)
- Correct backend service names and ports
- Proper rules and hosts defined

3. Check Service and Endpoints

Verify the backend service is working and has healthy endpoints:

kubectl get svc <service-name>
kubectl get endpoints <service-name>

If no endpoints, your service pods might not be running or labels/selectors might be wrong.

4. Check Logs of Ingress Controller Pod(s)

Look for errors or warning logs in ingress controller pods:

kubectl logs -n ingress-nginx <pod-name>
If multiple pods, check all or at least one pod to see the ingress controller logs.

5. Verify LoadBalancer Service IP or DNS

Get external IP or hostname of ingress controller's LoadBalancer service:

kubectl get svc -n ingress-nginx

Make sure this IP/DNS matches what you're trying to access via browser or curl.

6. Curl Test to Ingress Endpoint

Try hitting the ingress endpoint with curl and see response or errors:

curl -v http://<external-ip-or-hostname>/

This helps verify connectivity and response status.

7. Check DNS / /etc/hosts

- Ensure your domain resolves to the ingress controller external IP.
- Use nslookup <domain> or ping <domain> to verify DNS.
- For local testing, update /etc/hosts correctly.

8. Check Ingress Class

Verify the ingress resource is using the correct ingress class expected by the ingress controller:

kubectl get ingress <ingress-name> -o yaml | grep ingressClassName

Make sure it matches your controller's ingress class, e.g., nginx.

9. Validate Annotations

Some ingress controllers require specific annotations.

- Check if you have required annotations for rewrite, SSL redirect, etc.
- Verify if any annotation is causing a conflict.

10. Check Network Policies / Security Groups

- Confirm there are no Kubernetes NetworkPolicies blocking traffic to the ingress or backend pods.
- In AWS, check Security Groups allow inbound traffic on required ports (80/443).

11. Describe Pods and Services

Look for events or status issues on backend pods:

kubectl describe pod <pod-name>
kubectl describe svc <service-name>

12.Use kubectl proxy to test internally

Test the service inside the cluster:

kubectl port-forward svc/<service-name> 8080:80
curl http://localhost:8080

If this works but ingress doesn't, problem is likely in ingress config.

Conclusion

Ingress is a powerful Kubernetes resource that manages external access to services using simple routing rules. With an Ingress Controller like NGINX

(deployed via KOPS on AWS), you can efficiently route HTTP/HTTPS traffic using host-based, path-based, and TLS-secured configurations.

By practicing real-world YAML examples and learning how to debug common issues, you gain the skills needed to manage production-ready traffic in a Kubernetes environment.

Ingress simplifies service exposure, enhances security, and supports scalable application architecture.