Experiment: Ingress with Kind

We can leverage KIND's `extraPortMapping` config option when creating a cluster to forward ports from the host to an ingress controller running on a node.

We can also setup a custom node label by using `node-labels` in the kubeadm `InitConfiguration`, to be used by the ingress controller `nodeSelector`.

- Create a cluster
- Deploy the Ambassador Ingress controller
- Create the echo hello world services
- Create the ingress
- Annotate the ingress for Ambassador

Create Cluster

Create a kind cluster with `extraPortMappings` and `node-labels`.

- **`extraPortMappings`** allow the local host to make requests to the Ingress controller over ports 80/443
- **`node-labels`** only allow the ingress controller to run on a specific node(s) matching the label selector Ambassador will be installed with the help of the Ambassador operator.

First install the CRDs with

For MacOS

Start a terminal and cd to ~/Projects/kind

For Windows

Run Git Bash from the c:\Projects\kind folder

For MacOS and Windows

Create a kind cluster with a here document for the cluster config.

```
cat <<EOF | kind create cluster --name ambassador-test --image
kindest/node:v1.21.12 --config=-
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
```

```
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
EOF
```

```
Γ£ô Preparing nodes ≡ƒôª Creating cluster "ambassador-test" ...
 ΓÇó Ensuring node image (kindest/node:v1.21.1) ≡ƒû╝  ...
 Γ£ô Ensuring node image (kindest/node:v1.21.1) ≡ƒû╝
 ΓÇó Preparing nodes ≡ƒôª   ...
 Γ£ô Preparing nodes ≡ƒôª
 ΓÇó Writing configuration ≡ƒô£  ...
 Γ£ô Writing configuration ≡ƒô£
 ΓÇó Starting control-plane ≡ƒò╣∩╕Å  ...
 Γ£ô Starting control-plane ≡ƒò╣∩╕Å
 ΓÇó Installing CNI ≡ƒöî  ...
 Γ£ô Installing CNI ≡ƒöî
 ΓÇó Installing StorageClass ≡ƒÆ╛  ...
 Γ£ô Installing StorageClass ≡ƒÆ╛
Set kubectl context to "kind-ambassador-test"
You can now use your cluster with:

kubectl cluster-info --context kind-ambassador-test

Have a nice day! ≡ƒæï
```

$ kind get clusters

Remember that for **kubectl cluster-info** command, we need to prepend our cluster name with **kind-**

$ kubectl cluster-info --context kind-ambassador-test

Similarly unless we used the default cluster name “kind” we need to pass in **--name** for invocations to the kind cli.

$ **kind get kubeconfig**

```
ERROR: could not locate any control plane nodes
```

$ kind get kubeconfig --name ambassador-test

```
apiVersion: v1
clusters:
. . .
```

Ambassador

Ambassador will be installed with the help of the Ambassador operator.

Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. Operators follow Kubernetes principles, notably the control loop.

The Operator pattern aims to capture the key aim of a human operator who is managing a service or set of services. Human operators who look after specific applications and services have deep knowledge of how the system ought to behave, how to deploy it, and how to react if there are problems.

People who run workloads on Kubernetes often like to use automation to take care of repeatable tasks. The Operator pattern captures how you can write code to automate a task beyond what Kubernetes itself provides.

The most common way to deploy an Operator is to add the Custom Resource Definition (CRD) and its associated Controller to your cluster. The Controller will normally run outside of the control plane, much as you would run any containerized application. For example, you can run the controller in your cluster as a Deployment.

First install the CRDs with

$ kubectl apply -f https://github.com/datawire/ambassador-operator/releases/latest/download/ambassador-operator-crds.yaml

Now install the kind-specific manifest for installing Ambassador with the operator in the `ambassador` namespace:

$ **kubectl apply -n ambassador -f https://github.com/datawire/ambassador-operator/releases/latest/download/ambassador-operator-kind.yaml**

$ **kubectl wait --timeout=180s -n ambassador --for=condition=deployed ambassadorinstallations/ambassador**

```
ambassadorinstallation.getambassador.io/ambassador condition met
```

Ambassador is now ready for use. Now we'll define an ingress rule to use Ambassador.

Create Simple Echo Service

Use the **echo.yaml** file you'll find in the GitHub repo. This will create two simple services hello and howdy

$ **kubectl apply -f echo.yaml**

```
pod/hello-app created
service/hello-service created
pod/howdy-app created
service/howdy-service created
ingress.networking.k8s.io/echo-ingress created
```

```
kubernetes@DESKTOP-1M2VN7E MINGW64 /c/projects/kind
```
$ curl localhost/hello

```
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:--
--:--:--     0
```

```
kubernetes@DESKTOP-1M2VN7E MINGW64 /c/projects/kind
```
$ wget localhost/howdy

```
--2021-07-26 06:16:25--  http://localhost/howdy
Resolving localhost (localhost)... ::1, 127.0.0.1
Connecting to localhost (localhost)|::1|:80... connected.
HTTP request sent, awaiting response... 404 Not Found
2021-07-26 06:16:25 ERROR 404: Not Found.
```

Notice that we're not getting our expected response

Ambassador did not automatically load the `Ingress` defined there. `Ingress` resources must include the annotation `kubernetes.io/ingress.class: ambassador` for being recognized by Ambassador (otherwise they are just ignored). To add that annotation we will do the following.

```
$ kubectl annotate ingress echo-ingress
  kubernetes.io/ingress.class=ambassador
```

```
Now port forward from our local to the container port by using the
  following
```

```
kubectl port-forward pods/hello-app 8080:5678 &
kubectl port-forward pods/howdy-app 8081:5678 &
```

Test our services

$ curl -s localhost:8081/howdy
howdy

$ curl -s localhost:8080/hello
hello

Review the headers returned from the services

$ curl -I localhost/howdy

```
HTTP/1.1 200 OK
x-app-name: http-echo
x-app-version: 0.2.3
date: Mon, 26 Jul 2021 11:31:33 GMT
content-length: 6
content-type: text/plain; charset=utf-8
x-envoy-upstream-service-time: 0
server: envoy
```

Experiment Cleanup

Remove the services and ingress

$ kubectl delete -f echo.yaml

```
pod "hello-app" deleted
service "hello-service" deleted
pod "howdy-app" deleted
service "howdy-service" deleted
ingress.networking.k8s.io "echo-ingress" deleted
```

Delete the cluster

$ kind delete cluster --name ambassador-test

```
Deleting cluster "ambassador-test" ...
```