Creating Ingress Resources Based on Paths

In this lesson, first, we will look into the definition of an Ingress resource and then create the objects based on this definition.

Defining an Ingress Resource

We'll try to make our `go-demo-2-api` Service available through the port `80`. We'll do that by defining an Ingress resource with the rule to forward all requests with the path starting with `/demo` to the Service `go-demo-2-api`.

Looking into the Definition

Let's take a look at the Ingress' YAML definition.

```
cat ingress/go-demo-2-ingress.yml
```

The **output** is as follows.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: go-demo-2
  annotations:
    kubernetes.io/ingress.class: "nginx"
    ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
```

```
  - http:
    paths:
    - path: /demo

      backend:
        serviceName: go-demo-2-api
        servicePort: 8080
```

- **Line 5:** This time, `metadata` contains a field we haven't used before. The `annotations` section allows us to provide additional information to the Ingress Controller. As you'll see soon, Ingress API specification is concise and limited. That is done on purpose. The specification API defines only the fields that are mandatory for all Ingress Controllers. All the additional info an Ingress Controller needs is specified through `annotations`. That way, the community behind the Controllers can progress at great speed, while still providing basic general compatibility and standards.

> **i** The list of general annotations and the Controllers that support them can be found in the Ingress Annotations page. For those specific to the NGINX Ingress Controller, please visit the NGINX Annotations page, and for those specific to GCE Ingress, visit the ingress-gce page.

> ⚠ You'll notice that documentation uses `nginx.ingress.kubernetes.io/` annotation prefixes. That is a relatively recent change that, at the time of this writing, applies to the beta versions of the Controller. We're combining it with `ingress.kubernetes.io/` prefixes so that the definitions work in all Kubernetes versions.

- **Line 8:** We specified the annotation `nginx.ingress.kubernetes.io/ssl-redirect: "false"` which tells the Controller that we do NOT want to redirect all HTTP requests to HTTPS. We're forced to do so since we do not have SSL certificates for the exercises that follow.

Now that we shed some light on the `metadata` and `annotations`, we can move to the `ingress` specification.

- **Line 9-16:** We specified a set of `rules` in the `spec` section. They are used to configure Ingress resource. For now, our rule is based on `http` with a single `path` and a `backend`. All the requests with the `path` starting with

/demo will be forwarded to the service go-demo-2-api on the port 8080.

Creating the Resource

Now that we had a short tour around some of the Ingress configuration options, we can proceed and create the resource.

```
kubectl create \
    -f ingress/go-demo-2-ingress.yml

kubectl get \
    -f ingress/go-demo-2-ingress.yml
```

The **output** of the latter command is as follows.

```
NAME      HOSTS ADDRESS        PORTS AGE
go-demo-2 *     192.168.99.100 80    29s
```

We can see that the Ingress resource was created. Don't panic if, in your case, the address is blank. It might take a while for it to obtain it.

Let's see whether requests sent to the base path /demo work.

```
curl -i "http://$IP/demo/hello"
```

The **output** is as follows.

```
HTTP/1.1 200 OK
Server: nginx/1.13.5
Date: Sun, 24 Dec 2017 14:19:04 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 14
Connection: keep-alive

hello, world!
```

The status code 200 OK is a clear indication that this time, the application is accessible through the port 80. If that's not enough of assurance, you can observe the hello, world! response as well.

The go-demo-2 Service we're currently using is no longer properly configured for our Ingress setup. Using type: NodePort, it is configured to export the port 8080 on all of the nodes. Since we're expecting users to access the application

8080 on all of the nodes. Since we're expecting users to access the application

through the Ingress Controller on port 80, there's probably no need to allow external access through the port 8080 as well.

We should switch to the ClusterIP type. That will allow direct access to the Service only within the cluster, thus limiting all external communication through Ingress.

Deleting the Objects

We cannot just update the Service with a new definition. Once a Service port is exposed, it cannot be un-exposed. We'll delete the go-demo-2 objects we created and start over. Besides the need to change the Service type, that will give us an opportunity to unify everything in a single YAML file.

```
kubectl delete \
    -f ingress/go-demo-2-ingress.yml

kubectl delete \
    -f ingress/go-demo-2-deploy.yml
```

Creating First Resource Using Unified YAML

We removed the objects related to go-demo-2, and now we can take a look at the unified definition.

```
cat ingress/go-demo-2.yml
```

We won't go into details of the new definition since it does not have any significant changes. It combines ingress/go-demo-2-ingress.yml and ingress/go-demo-2-deploy.yml into a single file, and it removes type: NodePort from the go-demo-2 Service.

```
kubectl create \
    -f ingress/go-demo-2.yml \
    --record --save-config

curl -i "http://$IP/demo/hello"
```

We created the objects from the unified definition and sent a request to validate that everything works as expected. The response should be `200 OK` indicating that everything (still) works as expected.

Please note that Kubernetes needs a few seconds until all the objects are running as expected. If you were too fast, you might have received the response `404 Not Found` or `503` instead of `200 OK`. If that was the case, all you have to do is send the `curl` request again.

In the next lesson, we will go through the sequential break down of the Ingress resource's creation process.