In this lecture we will discuss about Ingress in Kubernetes. We will start with a simple scenario. You are deploying an application on Kubernetes for a company that has an online store selling products.

Your application would be available at say, my-online-store.com. You build the application into a docker image and deploy it on the Kubernetes cluster as a pod in a deployment.

Your application needs a database so you deploy a MySQL database as a pod and create a service of type cluster IP called My SQL Service to make it accessible to our application. Your application is now working.

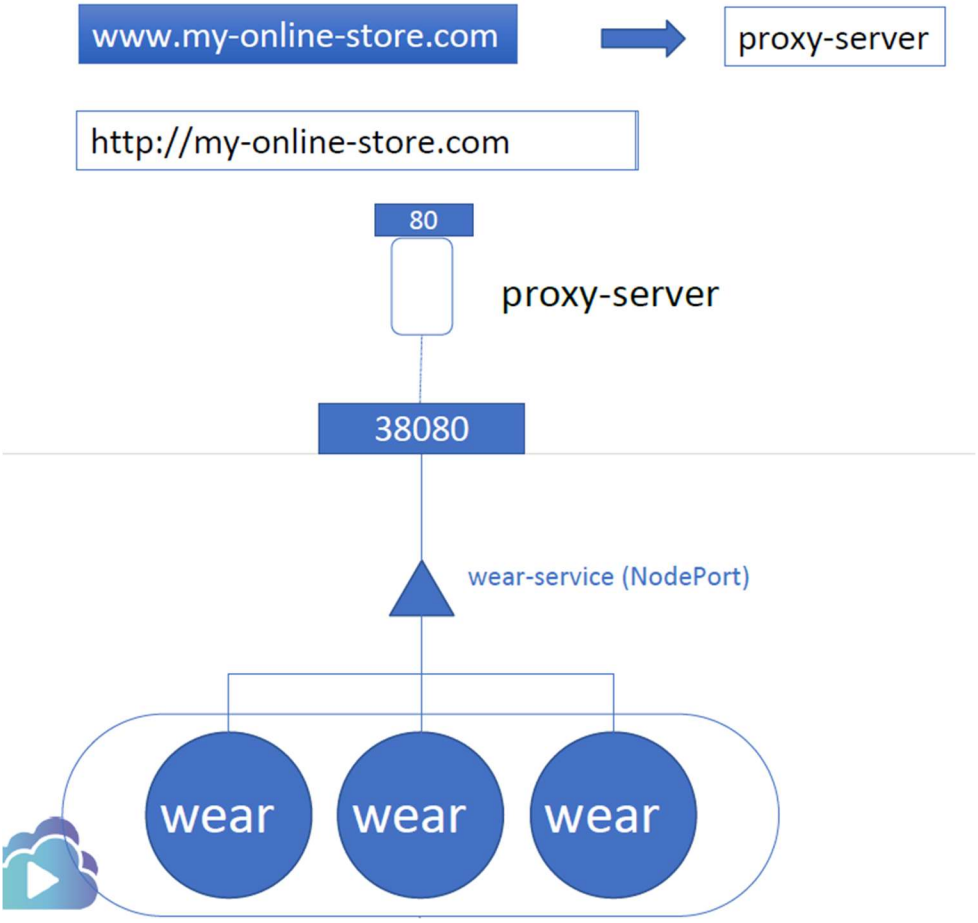
To make the application accessible to the outside world, you create another service, this time of type NodePort and make your application available on a high port on the nodes in the cluster. In this example, a port 38080 is allocated for the service. The users can now access your application using the URL http:// ip of any of your nodes followed by port 38080. That setup works and users are able to access the application.

Whenever traffic increases, we increase the number of replicas of the pod to handle the additional traffic, and the service takes care of splitting traffic between the pods.

However, if you have deployed a production-grade application before, you know that there are many more things involved in addition to simply splitting the traffic between the pods.

For example, we do not want the users to have to type in the IP address every time. So you configure your DNS server to point to the IP of the nodes. Your users can now access your application using the URL, my-online-store.com and port 38080. Now, you don't want your users to have to remember port number either.

However, service node ports can only allocate high numbered ports which are greater than 30,000. So you then bring in an additional layer between the DNS server and your cluster like a proxy server that proxies request on port 80 to port 38080 on your nodes. You then point your DNS to this server and users can now access your application by simply visiting my-online-store.com

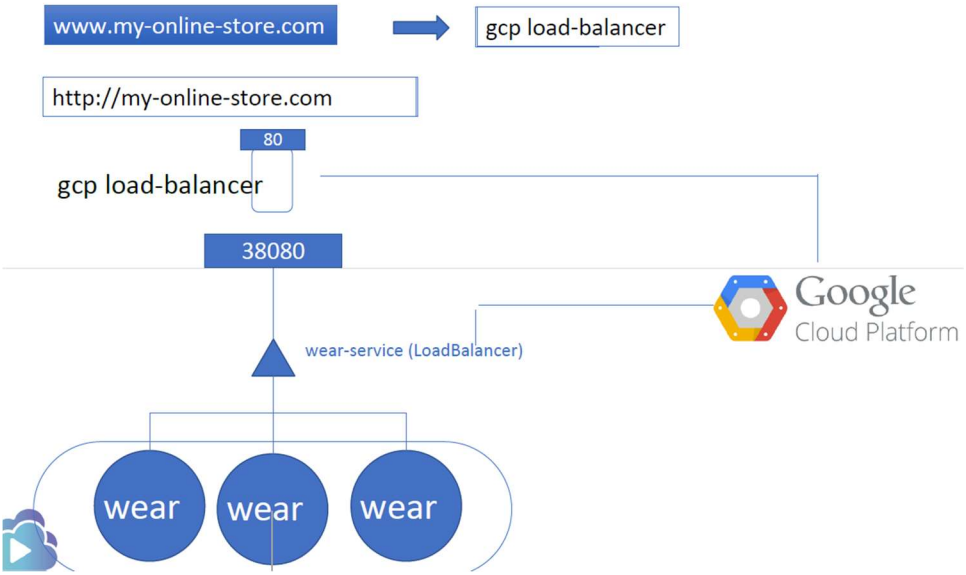


Now this is if your application is hosted on-prem in your data center.

Let's take a step back and see what you could do if you were on a public cloud environment like Google Cloud platform.

In that case, instead of creating a service of type node port for your wear application, you could set it to type load balancer. When you do that, Kubernetes would still do everything that it has to do for a node port which is to provision a high port for the service. But in addition to that, Kubernetes also sends a request to Google Cloud platform to provision a network load balancer for the service.

On receiving the request, GCP will then automatically deploy a load balancer configured to route traffic to the service ports on all nodes and return its information to Kubernetes. The load balancer has an external IP that can be provided to users to access the application. In this case, we set the DNS to point to this IP and users access the application using the URL, my-online-store.com. Perfect.

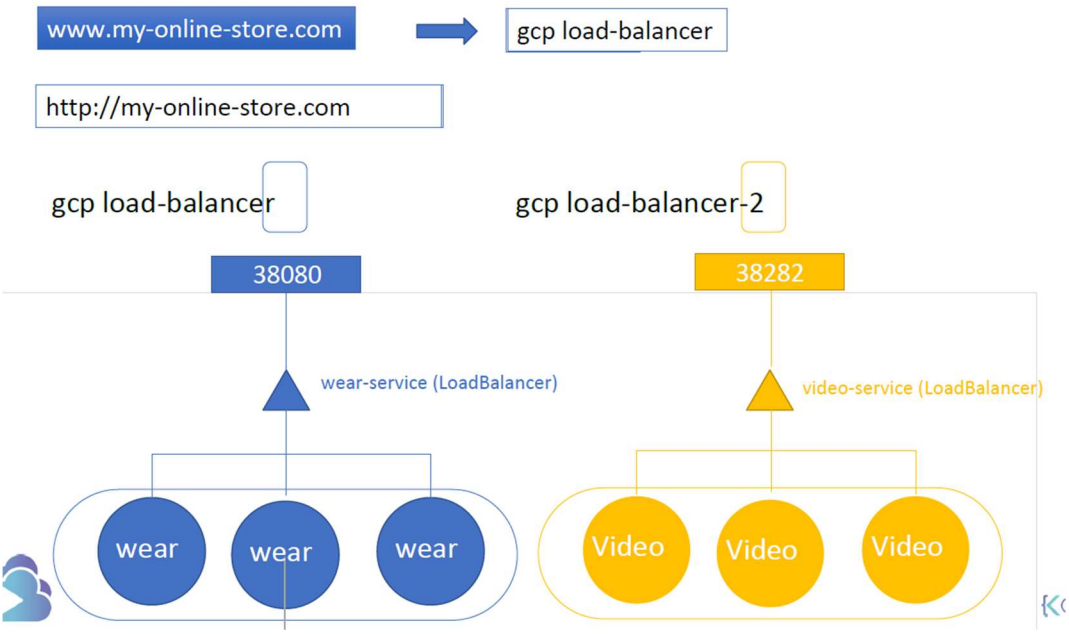


Your company's business grows and you now have new services for your customers. For example, a video streaming service. Now you want your users to be able to access your new video streaming service by going to my-online-store.com/watch.

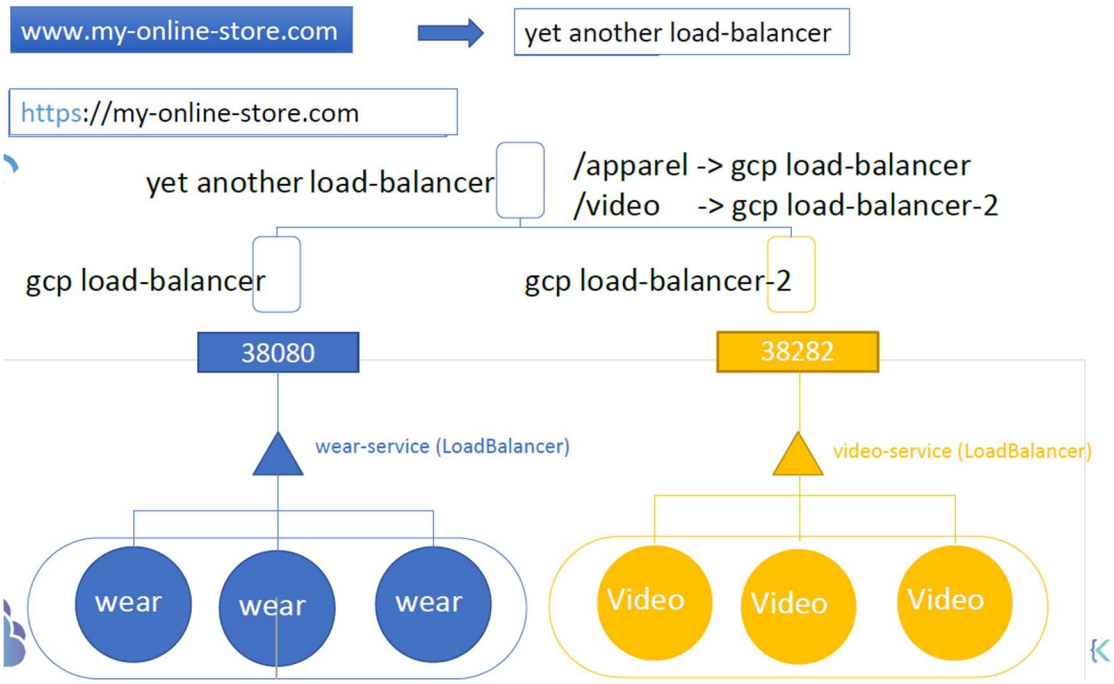
You'd like to make your old application accessible my-online-store.com/wear. Your developers developed the new video streaming application as a completely different application as it has nothing to do with the existing one.

However, to share the cluster's resources, you deploy the new application as a separate deployment within the same cluster. You create a service called video service of type load balancer. Kubernetes provision support 38282 for this service and also provisions a network load balancer on the cloud.

The new load balancer has a new ip. Remember, you must pay for each of these load balancers and having many such load balancers can inversely affect your cloud bill.



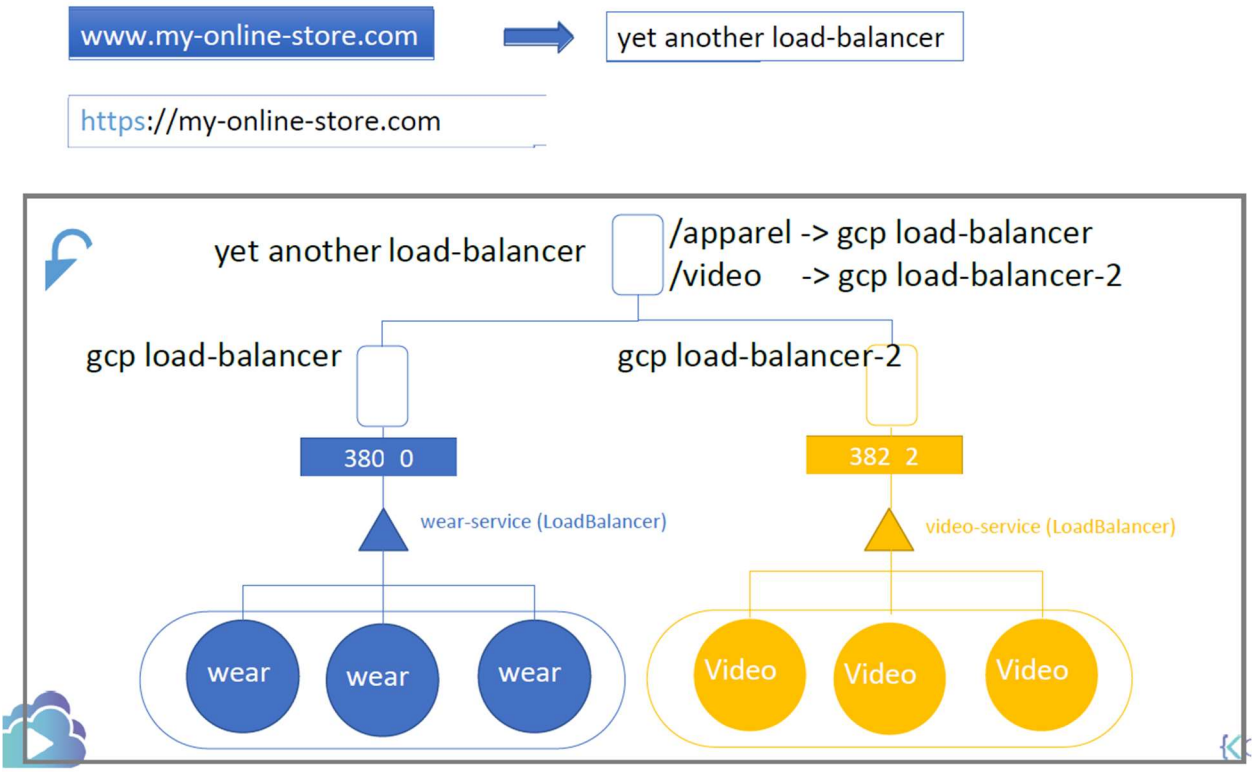
So how do you direct traffic between each of these load balancers based on the URL that the user types in? You need yet another proxy or load balancer that can redirect traffic based on URLs to the different services. Every time you introduce a new service, you have to reconfigure the load balancer.



So your users can access your application using https. Where do you configure that? It can be done at different levels, either at the application level itself, or at the load balancer level, or at the proxy server level, but which one? Now, you don't want your developers to implement it in their applications as they would do it in different ways, and it's an additional burden for them to develop additional code to handle that. You want it to be configured in one place with minimal maintenance.

Now, that's a lot of different configuration and all of these becomes difficult to manage when your application scales. It requires involving different individuals and different teams. You need to configure your firewall rules for each new service, and it's expensive as well as for each service, a new cloud-native load balancer needs to be provisioned.

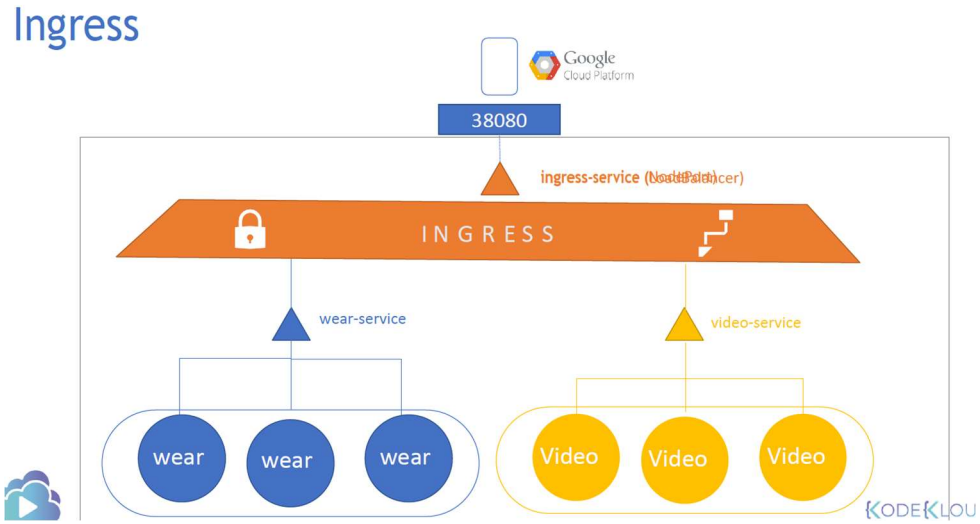
Wouldn't it be nice if you could manage all of that within the Kubernetes cluster and have all that configuration as just another Kubernetes definition file that lives along with the rest of your application deployment files? That's where Ingress comes in.



Ingres helps your users access your application using a single externally accessible URL that you can configure to route traffic to different services within your cluster based on the URL path. At the same time, implement SSL security as well.

Think of Ingress as a layer 7 load balancer built into the Kubernetes cluster that can be configured using native Kubernetes primitives just like any other object that we have been working with in Kubernetes.

Now, remember, even with Ingress you still need to expose it to make it accessible outside the cluster, so you still have to either publish it as a node port or with a cloud-native load balancer, but that is just a one-time configuration. Going forward, you're going to perform all your load balancing of SSL and URL-based routing configurations on the Ingress controller.



So how does it work? What is it? Where is it? How can you see it, and how can you configure it, and how does it load balance? How does it implement SSL?

Without Ingress, how would you do all of this? Well, I would use a reverse proxy or a load balancing solution like NGINX or HAproxy or Traefik. I would deploy them on my Kubernetes cluster and configure them to route traffic to other services. The configuration involves defining URL routes, configuring SSL certificates, et cetera.

Ingress is implemented by Kubernetes in kind of the same way. You first deploy a supported solution which happens to be any of these listed here, and then specify a set of rules to configure Ingress. The solution you deploy is called as an **Ingress controller**, and the set of rules you configure are called as **Ingress resources**. Ingress resources are created using definition files like the ones we have been using to create pods, deployments and services earlier in this course.

Now, remember, a Kubernetes cluster does not come with an Ingress controller by default. If you set up a cluster following the demos in this course, you won't have an Ingress controller built into it. So if you simply create Ingress resources and expect them to work, they won't.

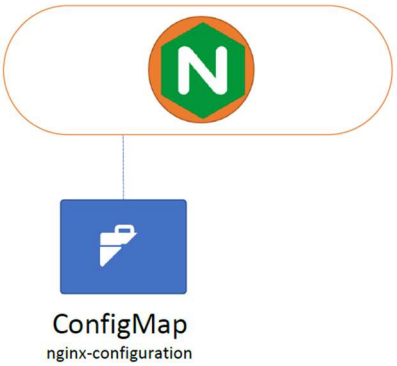
Let's look at each of these in a bit more detail. As I mentioned, you do not have an Ingress controller on Kubernetes by default, so you must deploy one. What do you deploy? There are a number of solutions available for Ingress, a few of them being GCE which is Google's layer 7 HTTP load balancer, NGINX, Contour, HAPROXY, Treafik, and Istio.

Out of this, GCE and NGINX are currently being supported and maintained by the Kubernetes project. And in this lecture we will use NGINX as an example.

These Ingress controller are not just another load balance or NGINX server. The load balancer components are just a part of it. Ingress controllers have additional intelligence built into them to monitor the Kubernetes cluster for new definitions or Ingress resources and configure the NGINX server accordingly.

An NGINX controller is deployed as just another deployment in Kubernetes. So we start with a deployment definition file named NGINX Ingress controller with one replica and a simple pod definition template. We will label it NGINX Ingress, and the image used is NGINX Ingress controller with the right version. This is a special built of NGINX, built specifically to be used as an Ingress controller in Kubernetes, so it has its own set of requirements.

Within the image, the NGINX program is stored at location /nginx-ingress-controller so you must pass that as the command to start the NGINX controller service. If you have worked with NGINX before, you know that it has a set of configuration options such as the path to store the logs, the keep alive threshold, SSL settings, session timeout, et cetera. In order to decouple this configuration data from the NGINX controller image, you must create a ConfigMap object and pass that in. Now remember, the ConfigMap object need not have any entries at this point. A blank object will do, but creating one makes it easy for you to modify a configuration setting in the future. You will just have to add it into this ConfigMap and not have to worry about modifying the NGINX configuration files.




```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
```

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx-ingress
  template:
    metadata:
      labels:
        name: nginx-ingress
    spec:
      containers:
        - name: nginx-ingress-controller
          image:quay.io/kubernetes-ingress-
                controller/nginx-ingress-controller:0.21.0
       args:
         - /nginx-ingress-controller
         - --configmap=$(POD_NAMESPACE)/nginx-configuration
```

You must also pass in two environment variables that carry the pod's name and name space it is deployed to. The NGINX service requires these to read the configuration data from within the pod. And finally, specify the ports used by the Ingress controller, which happens to be 80 and 443.

```
  env:
    - name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace

  ports:
    - name: http
      containerPort: 80
    - name: https
      containerPort: 443
```

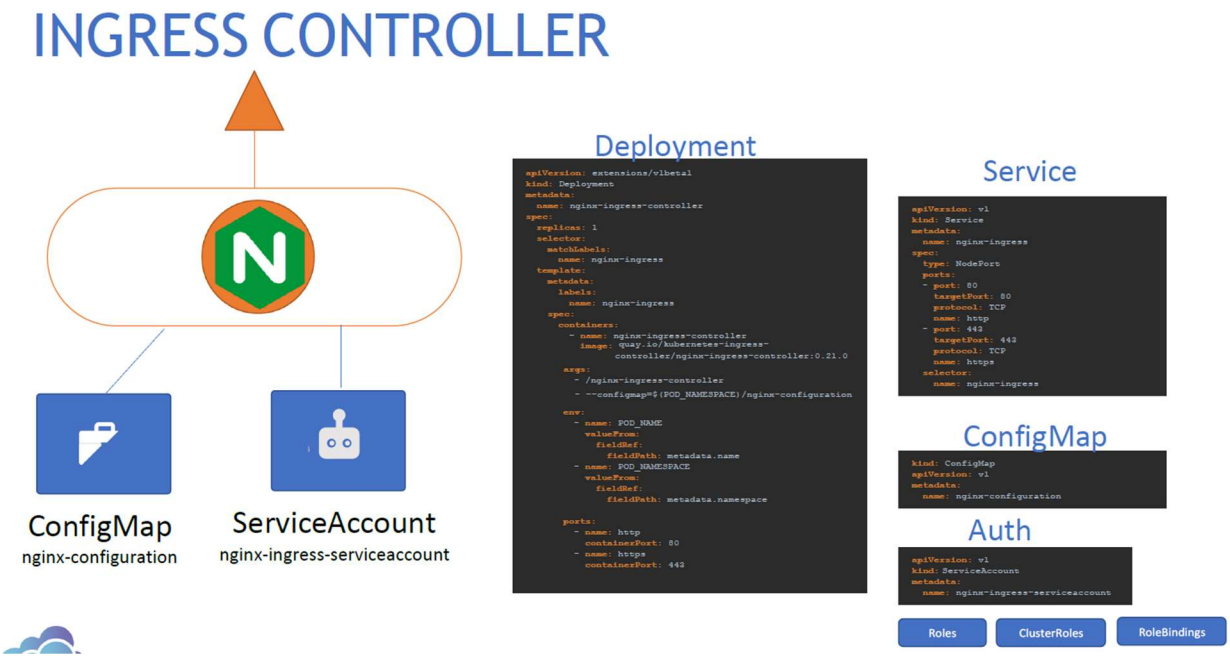
We then need a service to expose the Ingress controller to the external world, so we create a service of type node port with the NGINX Ingress label selector to link the service to the deployment.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    name: nginx-ingress
```

As mentioned before, the Ingress controllers have additional intelligence built into them to monitor the Kubernetes cluster for Ingress resources and configure the underlying ENGINX server when something is changed. But for the Ingress controller to do this, it requires a service account with the right set of permissions. For that, we create a service account with the correct Roles and RoleBindings.

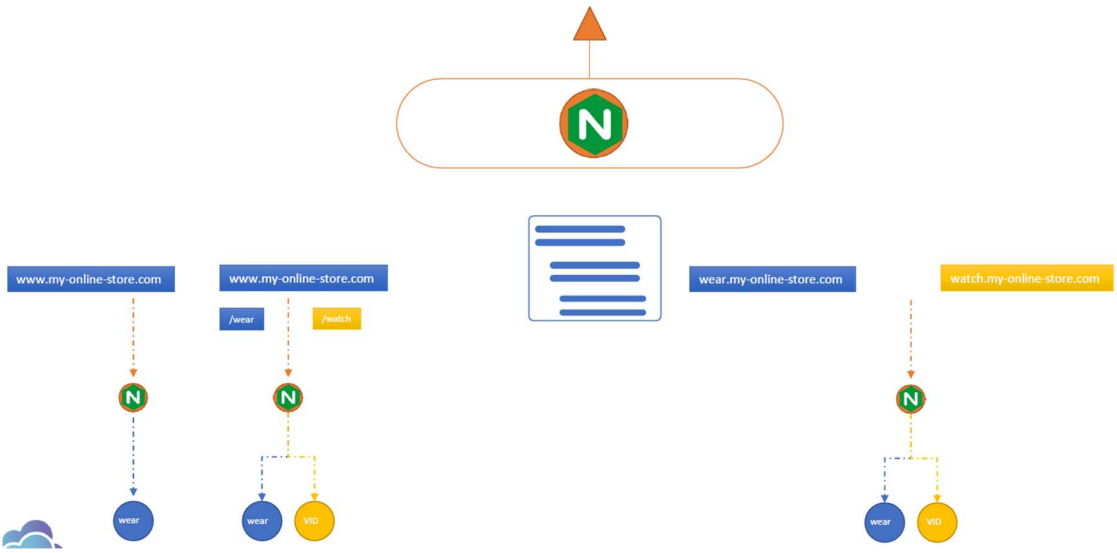
```yaml
apiVersion:  v1
kind: ServiceAccount
metadata:
  name: nginx-ingress-serviceaccount
```

So to summarize, with a deployment of the NGINX's Ingress image, a service to expose it, a ConfigMap to feed NGINX'S configuration data, and a service account with the right permissions to access all of these objects, we should be ready with an Ingress controller in its simplest form.



Now, onto the next part of creating Ingress resources. An Ingress resource is a set of rules and configurations applied on the Ingress controller. You can configure rules to say, simply forward all incoming traffic to a single application or route traffic to different applications based on the URL.

So if the user goes to my-online-store.com/wear, then route to one app, or if the user visits the watch URL, then route the user to the video app, or you could route user based on the domain name itself. For example, if the user visits wear.my-online-store.com, then route the user to the wear app, or else route the user to the video app.



Let us look at how to configure these in a bit more detail. The Ingress resource is created with a Kubernetes definition file. In this case, Ingress-wear.yaml. As with any other object, we have API version, kind, metadata and spec. The API version is extensions/v1beta1. kind is Ingress, and we will name it Ingress-wear, and under spec we have, backend. Now remember that the API version for Ingress is extensions/v1beta1 as of this recording, but this is expected to change with newer releases of Kubernetes.

So when you are deploying Ingress, always remember to refer to the Kubernetes Documentation to know exactly the right API version for that release of Kubernetes. So the traffic is, of course, routed to the application services and not pods directly. The backend section defines where the traffic will be routed to. So if it's a single backend, then you don't really have any rules. You can simply specify the service name and port of the backend wear-service.

Ingress-wear.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
 backend:
     serviceName: wear-service
     servicePort: 80
```

Create the Ingress resource by running the Kubectle create command. View the created Ingress by running the Kubectle get ingress command. The new Ingress is now created and routes all incoming traffic directly to the wear service.

```
kubectl create -f Ingress-wear.yaml
ingress.extensions/ingress-wear created

kubectl get ingress

ingress-wear        *                    80        2s
```

You use rules when you want to route traffic based on different conditions. For example, you create one rule for traffic originating from each domain or host name. That means when users reach your cluster using the domain name, my-online-store.com, you can handle that traffic using rule one. When users reach your cluster using domain name, wear.my-online-store.com, you can handle that traffic using a separate rule, rule two. Use rule three to handle traffic from watch.my-online-store.com and say, use the fourth rule to handle everything else.

And just in case you didn't know, you could get different domain names to reach your cluster by adding multiple DNS entries all pointing to the same Ingress controller service on your Kubernetes cluster.

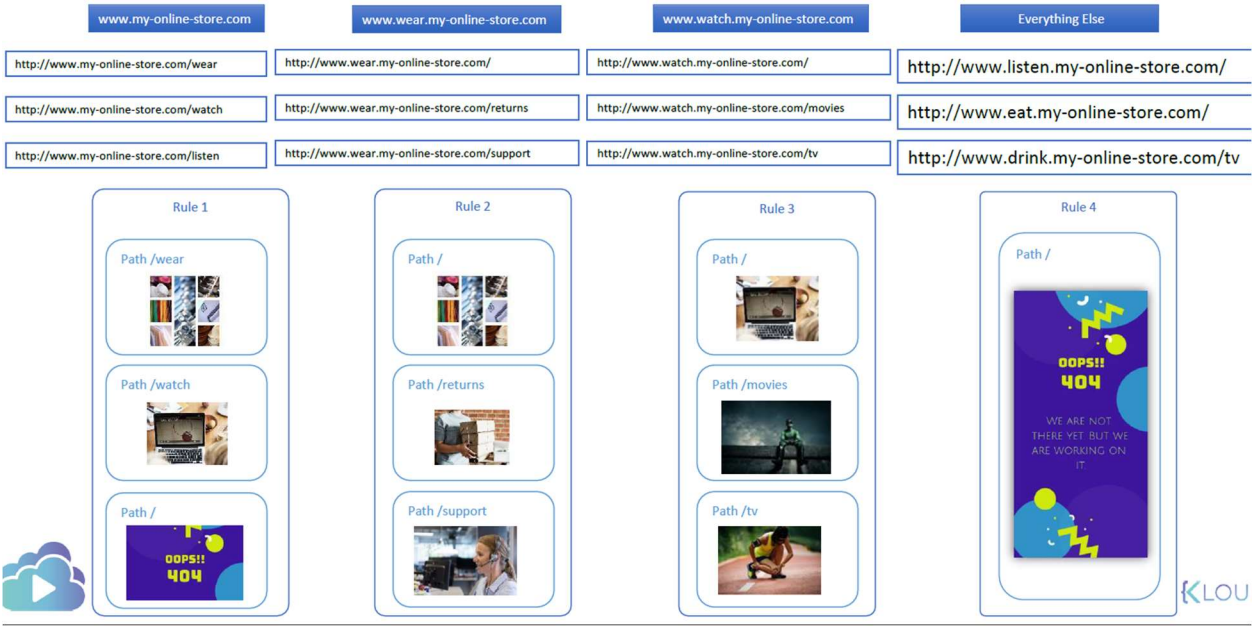
Now, within each rule, you can handle different paths. For example, within rule one, you can handle the wear path to route that traffic to the clothes application, and a watch path to route traffic to the video streaming application, and a third path that routes anything other than the first two to a 4 0 4 Not Found page.

Similarly, the second rule handles all traffic from wear.my-online-store.com. You can have path definition within this rule to route traffic based on different paths. For example, say you have different applications and services within the apparel section for shopping or returns or support. When a user goes to wear.my-online-store.com, by default, they reach the shopping page. But if they go to exchange or support URL, they reach a different backend service.

The same goes for rule three where you route traffic to watch.my-online-store.com to the video streaming application, but you can have additional paths in it such as movies or TV.

And finally, anything other than the ones listed here will go to the fourth rule. That would simply show a 4 0 4 Not Found Error page.

So remember, we have rules at the top for each host or domain name, and within each rule you have different paths to route traffic based on the URL.

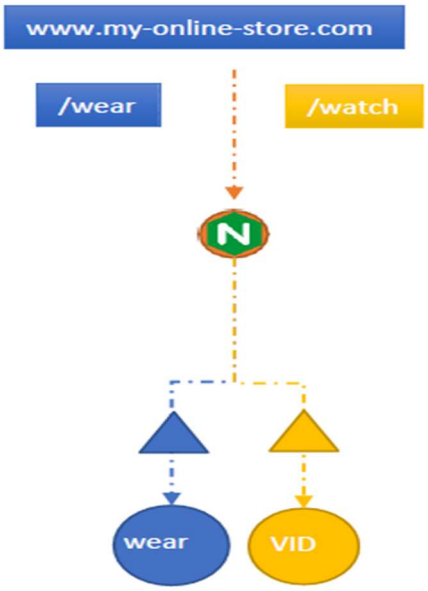


Now, let's look at how we configure Ingress resources in Kubernetes. We will start where we left off. We start with a similar definition file. This time under spec, we start with a set of rules. Now our requirement here is to handle all traffic coming into my-online-store.com and route them based on the URL path. So we just need a single URL for this since we are only handling traffic to a single domain name, which is my-online-store.com, in this case. Under rules, we have one item which is an HTTP rule in which we specify different paths.

So paths is an array of multiple items, one path for each URL. Then we move the backend we used in the first example under the first path. The backend specification remains the same, it has a service name and service port. Similarly, we create a similar backend entry to the second URL path for the watch service to route all traffic coming in through the watch URL to the watch service.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
  backend:
    serviceName: wear-service
    servicePort: 80
```

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
      paths:
      - path: /wear
        backend:
          serviceName: wear-service
          servicePort: 80
      - path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80
```

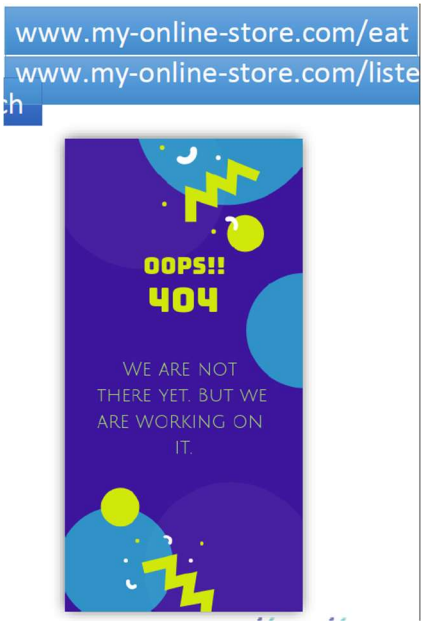


Create the Ingress resource using the Kubectl create command. Once created, view additional details about the Ingress resource by running the Kubectl describe Ingress command. You now see two backend URLs under the rules, and the backend service they are appointing to just as we created it.

```
kubectl describe ingress ingress-wear-watch
Name:             ingress-wear-watch
Namespace:        default
Address:
Default backend:  default-http-backend:80 (<none>)
Rules:
  Host  Path  Backends
  ----  ----  --------
  *
        /wear   wear-service:80 (<none>)
        /watch  watch-service:80 (<none>)
Annotations:
Events:
  Type    Reason  Age   From                      Message
  ----    ------  ----  ----                      -------
  Normal  CREATE  14s   nginx-ingress-controller  Ingress default/ingress-wear-watch
```

Now, if you look closely in the output of this command, you see that there is something about a default backend. Hmm, what might that be? If a user tries to access a URL that does not match any of these rules, then the user is directed to the service specified as the default backend. In this case, it happens to be a service named, default-http-backend. So you must remember to deploy as such a service.

Back in your application, say a user visits the URL, my-online-store.com/listen or eat, and you don't have an audio streaming or a food delivery service, you might wanna show them a nice message. You can do this by configuring a default backend service to display this 4 0 4 Not Found Error page.



The third type of configuration is using domain names or host names. We start by creating a similar definition file for Ingress. Now that we have two domain names, we create two rules, one for each domain. To split traffic by domain name, we use the host field. The host field in each rule matches the specified value with the domain name used in the request URL and routes traffic

to the appropriate backend. In this case, note that we only have a single backend path for each rule, which is fine. All traffic from these domain names will be routed to the appropriate backend irrespective of the URL path used. You can still have multiple path specifications in each of these to handle different URL paths.

Ingress-wear-watch.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:

  rules:
  - host: wear.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: wear-service
          servicePort: 80
  - host: watch.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: watch-service
          servicePort: 80
```

Now, let's compare the two. Splitting traffic by URL had just one rule and we split the traffic with two paths. To split traffic by host name, we used two rules and one path specification in each rule.

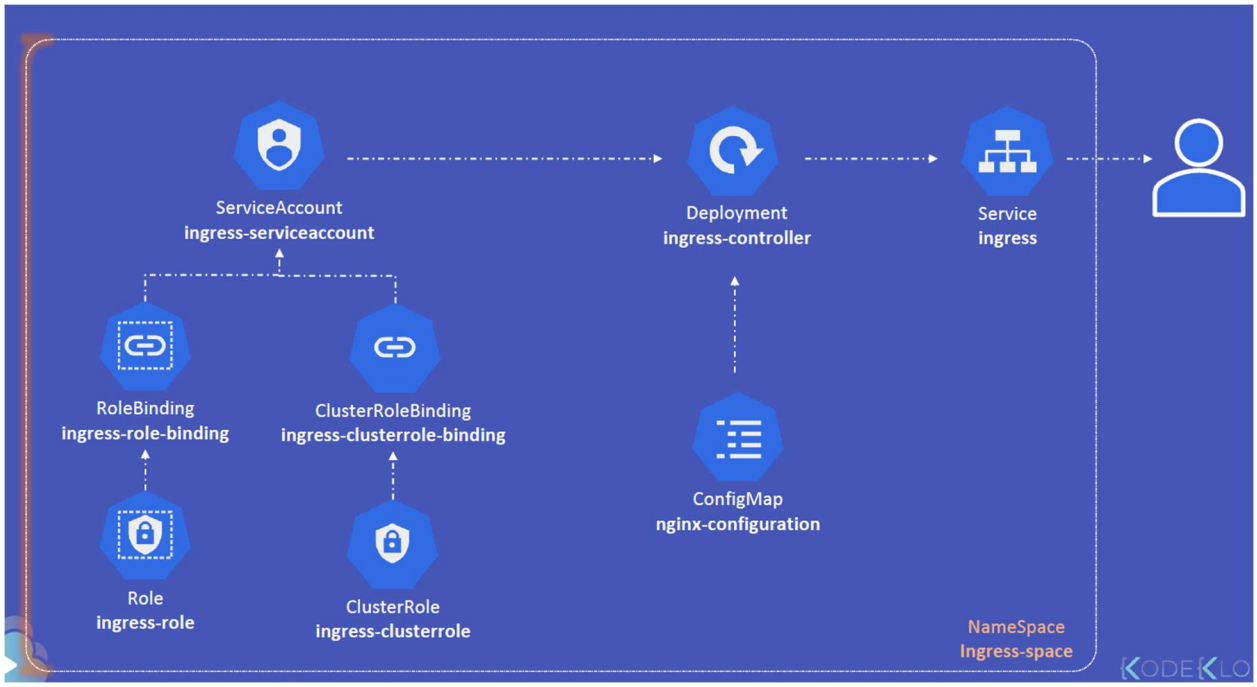
Ingress-wear-watch.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
      paths:
      - path: /wear
        backend:
          serviceName: wear-service
          servicePort: 80
      - path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80
```

Ingress-wear-watch.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:

  rules:
  - host: wear.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: wear-service
          servicePort: 80
  - host: watch.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: watch-service
          servicePort: 80
```


ServiceAccount
ingress-serviceaccount
Deployment
ingress-controller
Service
ingress
RoleBinding
ingress-role-binding
ClusterRoleBinding
ingress-clusterrole-binding
ConfigMap
nginx-configuration
Role
ingress-role
ClusterRole
ingress-clusterrole
NameSpace
Ingress-space
KODEKLO