**Getting Started with Ingress**

Learn what is ingress and why it should be used.

**We'll cover the following**

* [Why Use Ingress Objects?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N732V7qGWo2#Why-Use-Ingress-Objects?)

**Why Use Ingress Objects?**

Applications that are not accessible to users are useless. Kubernetes Services provide accessibility with a usability cost. Each application can be reached through a different port. We **cannot** expect users to know the port of each service in our cluster.

Ingress objects manage external access to the applications running inside a Kubernetes cluster.

While, at first glance, it might seem that we already accomplished that through Kubernetes Services, they do not make the applications truly accessible. We still need forwarding rules based on paths and domains, SSL termination and a number of other features.

In a more traditional setup, we’d probably use an external proxy and a load balancer. Ingress provides an API that allows us to accomplish these things, in addition to a few other features we expect from a dynamic cluster.

**Why Services Are Not the Best Fit for External Access?**

In this lesson, we will discover why services are not the best fit for enabling external access to the applications.

**We'll cover the following**

* [Only services won’t suffice](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAG2NkOLjPG#Only-services-won%E2%80%99t-suffice)
* [Access through services](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAG2NkOLjPG#Access-through-services)
* [Understanding the process](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAG2NkOLjPG#Understanding-the-process)
* [The solution](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAG2NkOLjPG#The-solution)
* [SSL certificates](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAG2NkOLjPG#SSL-certificates)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAG2NkOLjPG#Try-it-yourself)
* [Troubleshooting with minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAG2NkOLjPG#Troubleshooting-with-minikube)

**Only services won’t suffice**

We cannot explore solutions before we know what the problems are. Therefore, we’ll re-create a few objects using the knowledge we already gained. That will let us see whether Kubernetes Services satisfy all the needs users of our applications might have. Or, to be more explicit, we’ll explore which features we’re missing when making our applications accessible to users.

We already discussed that it is a bad practice to publish fixed ports through Services. That method is likely to result in conflicts or, at the very least, create the additional burden of carefully keeping track of which port belongs to which Service. We already discarded that option before, and we won’t change our minds now.

kubectl create -f go-demo-2-deploy.yml

kubectl get -f go-demo-2-deploy.yml

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

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/go-demo-2-db 0/1 1 0 2m15s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/go-demo-2-db ClusterIP 10.111.211.179 <none> 27017/TCP 2m15s

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/go-demo-2-api 0/3 3 0 2m15s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/go-demo-2-api NodePort 10.103.180.226 <none> 8080:30753/TCP 2m15s

As you can see, these are the same Services and Deployments we previously created.

Before we move on, we should wait until all the Pods are up and running.

kubectl get pods

The **output** is as follows.

NAME READY STATUS RESTARTS AGE

go-demo-2-api-68df567fb5-8qcmv 1/1 Running 0 3m

go-demo-2-api-68df567fb5-k55d4 1/1 Running 0 3m

go-demo-2-api-68df567fb5-ws9cj 1/1 Running 0 3m

go-demo-2-db-dd48b7dfc-hdxbz 1/1 Running 0 3m

If, in your case, some of the Pods are not yet running, please wait a few moments and re-execute the kubectl get pods command. We’ll continue once they’re ready.

## Access through services

One obvious way to access the applications is through Services. Since the service go-demo-2-api in go-demo-2-deploy.yml is listening on port 8080 while the platform is listening on port 3000 thus we are binding the port while calling the service. We used that information to send a request.

kubectl port-forward service/go-demo-2-api 3000:8080 --address 0.0.0.0

#Now open a new terminal and run the following command

curl -i "0.0.0.0:3000/demo/hello"

#close the new terminal after running and observing the output of curl

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

HTTP/1.1 200 OK

Date: Sun, 24 Dec 2017 13:35:26 GMT

Content-Length: 14

Content-Type: text/plain; charset=utf-8

hello, world!

The application responded with the status code "200" thus confirming that the Service indeed forwards the requests.

While publishing a random, or even a hard-coded port of a single application might not be so bad, if we’d apply the same principle to more applications, the user experience would be horrible. To make the point a bit clearer, we’ll deploy another application.

kubectl create -f devops-toolkit-dep.yml --record --save-config

kubectl get -f devops-toolkit-dep.yml

This application follows similar logic to the first. From the latter command, we can see that it contains a Deployment and a Service. The details are of no importance since the YAML definition is very similar to those we used before. What matters is that now we have two applications running inside the cluster.

## Understanding the process

Let’s check whether the new application is indeed reachable.

kubectl port-forward service/devops-toolkit 3000:80 --address 0.0.0.0

#Now open a new terminal and run the following command

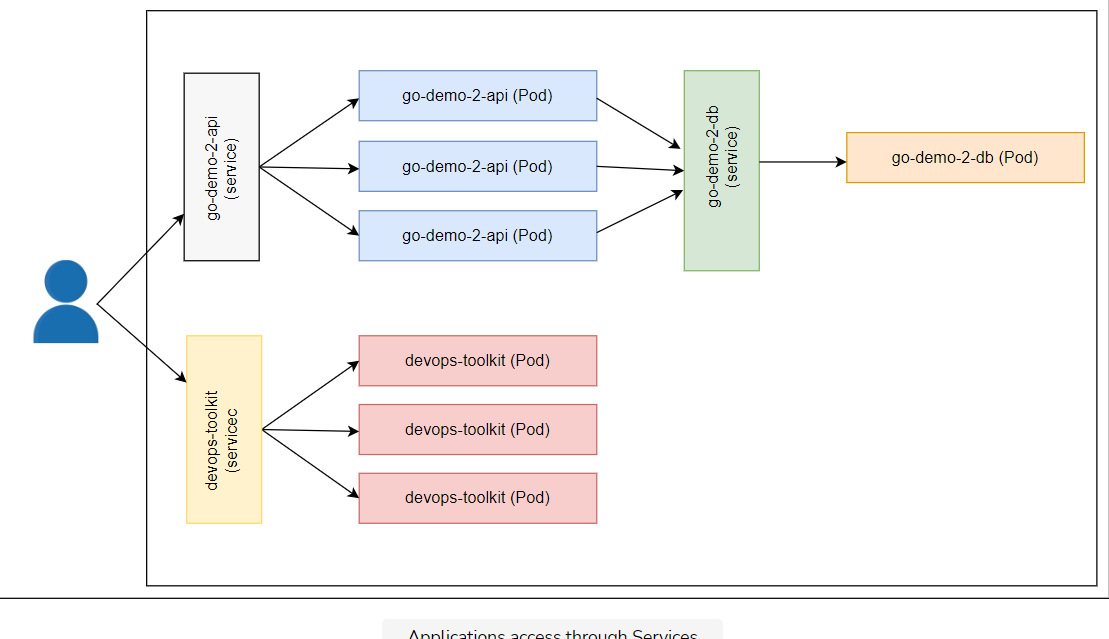
curl -i "0.0.0.0:3000"

#close the new terminal after running and observing the output of curl

We retrieved the port of the new Service and opened the application in a browser. If you get a page not found error, you might want to wait a bit longer until the containers are pulled, and try again

A simplified flow of requests is depicted in the below-given illustration.

A user sends a request to one of the nodes of the cluster. That request is received by a Service and load balanced to one of the associated Pods. It’s a bit more complicated than that, with iptables, kube DNS, kube proxy, and a few other things involved in the process. We explored them in more detail in the Using Services To Enable Communication Between Pods chapter, and there’s probably no need to go through them all again. For the sake of brevity, the simplified diagram should do.



We cannot expect our users to know specific ports behind each of those applications. Even with only two, that would not be very user-friendly. If that number would rise to tens or even hundreds of applications, our business would be very short-lived.

What we need is a way to make all services accessible through standard HTTP (80) or HTTPS (443) ports. Kubernetes Services alone cannot get us there. We need more.

## The solution

What we need is to grant access to our services on predefined paths and domains. Our go-demo-2 service could be distinguished from others through the base path /demo. Similarly, the books application could be reachable through the devopstoolkitseries.com domain. If we could accomplish that, we could access them with the commands as follows.

curl -i "0.0.0.0:80/demo/hello"

The request received the “default backend - 404” response. There is no process listening on port 80, so this outcome is not a surprise. We could have changed one of the Services to publish the fixed port 80 instead assigning a random one. Still, that would provide access only to one of the two applications.

We often want to associate each application with a different domain or sub-domain. Outside the examples we’re running, the books application is accessible through the [devopstoolkitseries.com](http://www.devopstoolkitseries.com/) domain. Since access to the domain is not feasible, we’ll simulate it by adding the domain to the Host header.

The command that should verify whether the application running inside our cluster is accessible through the devopstoolkitseries.com domain is as follows.

curl -i \

-H "Host: devopstoolkitseries.com" \

<http://0.0.0.0:80>

As expected, the request is still refused.

## SSL certificates

Last, but not the least, we should be able to make some, if not all, applications (partly) secure by enabling HTTPS access. That means that we should have a place to store our SSL certificates. We could put them inside our applications, but that would only increase the operational complexity. Instead, we should aim towards SSL offloading somewhere between clients and the applications, and it should come as no surprise that Kubernetes has a solution for all these.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create -f go-demo-2-deploy.yml

kubectl get -f go-demo-2-deploy.yml

kubectl get pods

kubectl port-forward service/go-demo-2-api 3000:8080 --address 0.0.0.0

#Now open a new terminal and run the following command

curl -i "0.0.0.0:3000/demo/hello"

#close the new terminal after running and observing the output of curl

kubectl create -f devops-toolkit-dep.yml --record --save-config

kubectl get -f devops-toolkit-dep.yml

kubectl port-forward service/devops-toolkit 3000:80 --address 0.0.0.0

#Now open a new terminal and run the following commands

curl -i "0.0.0.0:3000"

curl -i "0.0.0.0:80/demo/hello"

curl -i \

-H "Host: devopstoolkitseries.com" \

<http://0.0.0.0:80>

## Troubleshooting with minikube

For deploying the services and running API on a local machine using minikube, use the following code. Here IP will retrieve the IP of the minikube virtual machine while PORT will get the nodePort from the YAML file. The curl command will send an HTTP request to the service.

IP=$(minikube ip)

PORT=$(kubectl get svc go-demo-2-api \

-o jsonpath="{.spec.ports[0].nodePort}")

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

**Enabling Ingress Controllers**

Learn how to enable the Ingress controller and play around with it.

**We'll cover the following**

* [Why ingress controllers are required?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YMm9noR2nVM#Why-ingress-controllers-are-required?)
* [Enabling ingress on k3d](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YMm9noR2nVM#Enabling-ingress-on-k3d)
* [Playing Around with the Controller](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YMm9noR2nVM#Playing-Around-with-the-Controller)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YMm9noR2nVM#Try-it-yourself)

**Why ingress controllers are required?**

We need a mechanism that will accept requests on pre-defined ports (e.g., 80 and 443) and forward them to Kubernetes Services. It should be able to distinguish requests based on paths and domains as well as to be able to perform SSL offloading.

Kubernetes itself does not have a ready-to-go solution for this. Unlike other types of Controllers that are typically part of the kube-controller-manager binary, Ingress Controller needs to be installed separately. Instead of a Controller, kube-controller-manager offers *Ingress resource* that other third-party solutions can utilize to provide requests forwarding and SSL features. In other words, Kubernetes only provides an *API*, and we need to set up a Controller that will use it.

Fortunately, the community already built a myriad of Ingress Controllers. We won’t evaluate all of the available options since that would require a lot of space, and it would mostly depend on your needs and your hosting vendor.

**Enabling ingress on k3d**

Enable Ingress on k3d using the following command.

kubectl apply -f <https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.3.0/deploy/static/provider/cloud/deploy.yaml>

The **output** is as follows.

namespace/ingress-nginx created

serviceaccount/ingress-nginx created

serviceaccount/ingress-nginx-admission created

role.rbac.authorization.k8s.io/ingress-nginx created

role.rbac.authorization.k8s.io/ingress-nginx-admission created

clusterrole.rbac.authorization.k8s.io/ingress-nginx created

clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created

rolebinding.rbac.authorization.k8s.io/ingress-nginx created

rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created

clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created

clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created

configmap/ingress-nginx-controller created

service/ingress-nginx-controller created

service/ingress-nginx-controller-admission created

deployment.apps/ingress-nginx-controller created

job.batch/ingress-nginx-admission-create created

job.batch/ingress-nginx-admission-patch created

ingressclass.networking.k8s.io/nginx created

validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-nginx-admission created

Wait for the ingress controller to run using the following command.

kubectl get pods --namespace=ingress-nginx

ℹ️ In the rest of the lessons of this chapter the code playgrounds are preconfigured with the Ingress controller. However, it is suggested to please verify if the Ingress controller is running before using the service.

## Playing Around with the Controller

Now that the Ingress addon is enabled, we’ll check whether it is running inside our cluster.

kubectl get pods -n ingress-nginx \

| grep ingress

Ignore the -n argument. We did not yet explore Namespaces. For now, please note that the output of the command should show that “nginx-ingress-controller-…” Pod is running.

If the output is empty, you might need to wait for a few moments until the containers are pulled, and re-execute the kubectl get all --namespace ingress-nginx command again.

ℹ️ The ingress image is based on [NGINX Ingress Controller](https://github.com/kubernetes/ingress-nginx/blob/master/README.md). It is one of the only two currently supported and maintained by the Kubernetes community. The other one is [GLBC](https://github.com/kubernetes/ingress-gce/blob/master/README.md) that comes with [Google Compute Engine (GCE)](https://cloud.google.com/compute/) Kubernetes hosted solution.

By default, the Ingress Controller is configured with only two endpoints.

If we’d like to check Controller’s health, we can send a request to /healthz.

nohup kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 3000:80 --address 0.0.0.0 > /dev/null 2>&1 &

# Please wait for a few seconds before running the next command

curl -i "0.0.0.0:3000/healthz"

The **output** is as follows

HTTP/1.1 200 OK

Server: nginx/1.15.9

Date: Mon, 10 Jun 2019 12:02:11 GMT

Content-Type: text/html

Content-Length: 0

Connection: keep-alive

It responded with the status code “200 OK”, thus indicating that it is healthy and ready to serve requests. There’s not much more to it so we’ll move to the second endpoint.

The Ingress Controller has a default catch-all endpoint that is used when a request does not match any of the other criteria. Since we did not yet create any Ingress Resource, this endpoint should provide the same response to all requests except /healthz.

curl -i "0.0.0.0:3000/something"

The **output** is as follows.

HTTP/1.1 404 Not Found

Server: nginx/1.13.5

Date: Sun, 24 Dec 2017 15:36:23 GMT

Content-Type: text/plain; charset=utf-8

Content-Length: 21

Connection: keep-alive

default backend – 404

We got the response indicating that the requested resource could not be found.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.3.0/deploy/static/provider/cloud/deploy.yaml

kubectl get pods --namespace=ingress-nginx

kubectl get pods -n ingress-nginx \

| grep ingress

nohup kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 3000:80 --address 0.0.0.0 > /dev/null 2>&1 &

# Please wait for a few seconds before running the next commands

curl -i "0.0.0.0:3000/healthz"

curl -i "0.0.0.0:3000/something"

# Creating Ingress Resources Based on Paths

Understand the definition of an Ingress resource and then create the objects based on this definition.

**We'll cover the following**

* [Defining an Ingress resource](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R10yvPLQMVK#Defining-an-Ingress-resource)
  + [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R10yvPLQMVK#Looking-into-the-definition)
* [Creating the resource](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R10yvPLQMVK#Creating-the-resource)
* [Deleting the objects](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R10yvPLQMVK#Deleting-the-objects)
* [Creating first resource using unified YAML](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R10yvPLQMVK#Creating-first-resource-using-unified-YAML)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R10yvPLQMVK#Try-it-yourself)

## 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 go-demo-2-ingress.yml:

apiVersion: networking.k8s.io/v1

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

pathType: ImplementationSpecific

backend:

service:

name: go-demo-2-api

port:

number: 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.

ℹ️ The list of general annotations and the Controllers that support them can be found in the [Ingress Annotations](https://github.com/kubernetes/ingress-nginx/blob/master/docs/user-guide/nginx-configuration/annotations.md) page. For those specific to the [NGINX Ingress Controller](https://github.com/kubernetes/ingress-nginx/blob/master/README.md), please visit the [NGINX Annotations](https://github.com/kubernetes/ingress-nginx/blob/master/docs/user-guide/nginx-configuration/annotations.md) page, and for those specific to GCE Ingress, visit the [ingress-gce](https://github.com/kubernetes/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 9-19:** We specified a set of rules in the spec section. They are used to configure Ingress resources. 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 go-demo-2-ingress.yml

kubectl get -f 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.

nohup kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 3000:80 --address 0.0.0.0 > /dev/null 2>&1 &

curl -i <http://0.0.0.0:3000/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 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 go-demo-2-ingress.yml

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

kubectl create -f go-demo-2.yml \

--record --save-config

nohup kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 3000:80 --address 0.0.0.0 > /dev/null 2>&1 &

curl -i <http://0.0.0.0:3000/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.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create -f go-demo-2-ingress.yml

kubectl get -f go-demo-2-ingress.yml

nohup kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 3000:80 --address 0.0.0.0 > /dev/null 2>&1 &

curl -i "http://0.0.0.0:3000/demo/hello"

kubectl delete -f go-demo-2-ingress.yml

kubectl delete -f go-demo-2-deploy.yml

kubectl create -f go-demo-2.yml \

--record --save-config

nohup kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 3000:80 --address 0.0.0.0 > /dev/null 2>&1 &

curl -i <http://0.0.0.0:3000/demo/hello>

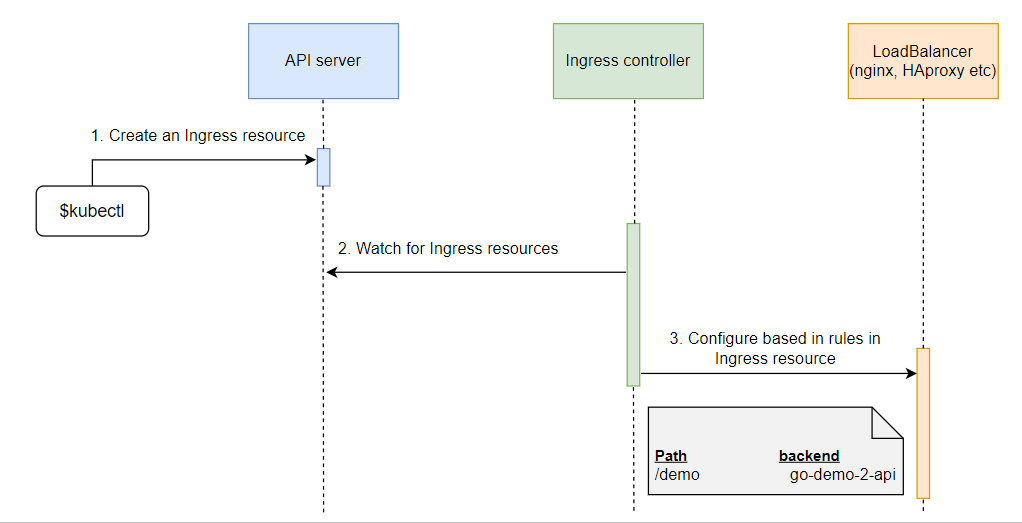
**Sequential Breakdown of the Process**

In this lesson, we will first go through the sequential breakdown of Ingress resource creation process and then create the second Ingress resource.

**We'll cover the following...**

Let’s see, through a sequence diagram, what happened when we created the Ingress resource.

1. The Kubernetes client (kubectl) sent a request to the API server requesting the creation of the Ingress resource defined in the go-demo-2.yml file.
2. The ingress controller is watching the API server for new events. It detected that there is a new Ingress resource.
3. The ingress controller configured the load balancer. In this case, it is nginx which modified nginx.conf with the values of all go-demo-2-api endpoints.



Now that one of the applications is accessible through Ingress, we should apply the same principles to the other.

## Creating Second Ingress Resource

Let’s first look into the definition and then create our second resource using devops-toolkit.yml.

### Looking into the Definition

Let’s take a look at the full definition of all the objects behind the devops-toolkit application.

The devops-toolkit Ingress resource is very similar to go-demo-2.

The only significant difference is that the path is set to /.

It will serve all requests. It would be a much better solution if we’d change it to a unique base path (e.g., /devops-toolkit) since that would provide a unique identifier.

However, this application does not have an option to define a base path, so an attempt to do so in Ingress would result in a failure to retrieve resources. We’d need to write rewrite rules instead. We could, for example, create a rule that rewrites path base /devops-toolkit to /.

That way if, for example, someone sends a request to /devops-toolkit/something, Ingress would rewrite it to /something before sending it to the destination Service. While such an action is often useful, we’ll ignore it for now. For now, / as the base path should do.

Apart from adding Ingress to the mix, the definition removed type: NodePort from the Service. This is the same type of action we did previously with the go-demo-2 service. We do not need external access to the Service.

### Deleting and Recreating the Objects

Let’s create the objects defined in the devops-toolkit.yml file.

kubectl create -f devops-toolkit.yml --record --save-config

Let’s take a look at the Ingresses running inside the cluster.

kubectl get ing

The **output** is as follows.

NAME HOSTS ADDRESS PORTS AGE

devops-toolkit \* 192.168.99.100 80 20s

go-demo-2 \* 192.168.99.100 80 58s

We can see that now we have multiple Ingress resources. The Ingress Controller (in this case NGINX) configured itself taking both of those resources into account.

We can define multiple Ingress resources that will configure a single Ingress Controller.

Let’s confirm that both applications are accessible through HTTP (port 80).

nohup kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 3000:80 --address 0.0.0.0 > /dev/null 2>&1 &

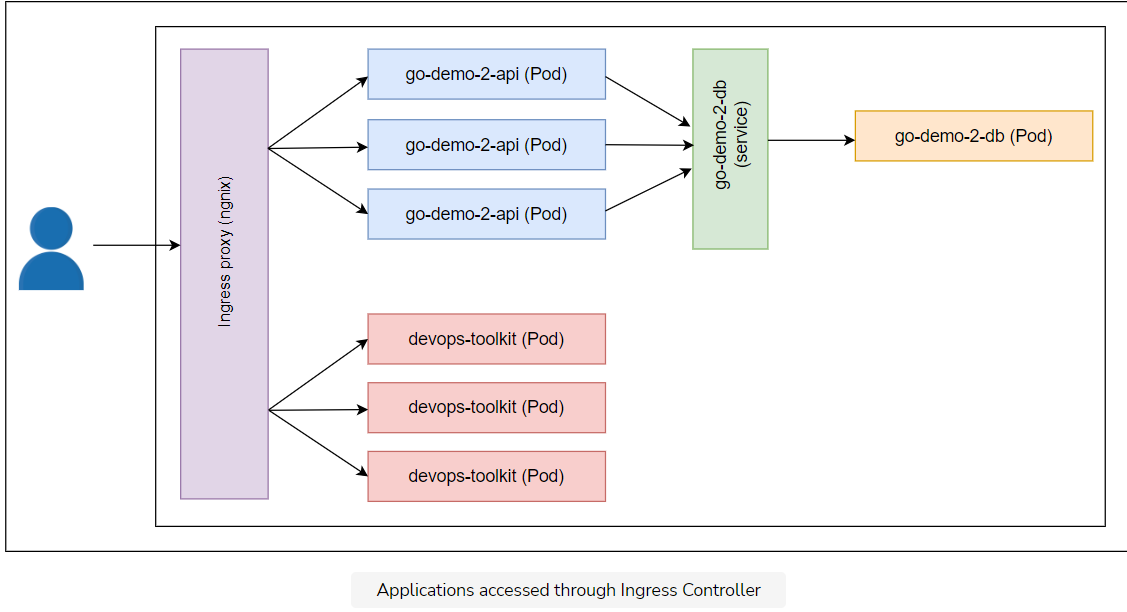
#Now click on the link beside the run button

#Next run the following command

curl <http://0.0.0.0:3000/demo/hello>

We’re able to view the application, whereas the curl command returned the already familiar hello, world! Message.

Ingress is a (kind of) Service that runs on all nodes of a cluster. A user can send requests to any and, as long as they match one of the rules, they will be forwarded to the appropriate Service.



Even though we can send requests to both applications using the same port (80), that is often a sub-optimal solution. Our users would probably be happier if they could access those applications through different domains.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create -f devops-toolkit.yml --record --save-config

kubectl get ing

nohup kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 3000:80 --address 0.0.0.0 > /dev/null 2>&1 &

curl "http://0.0.0.0:3000/demo/hello"