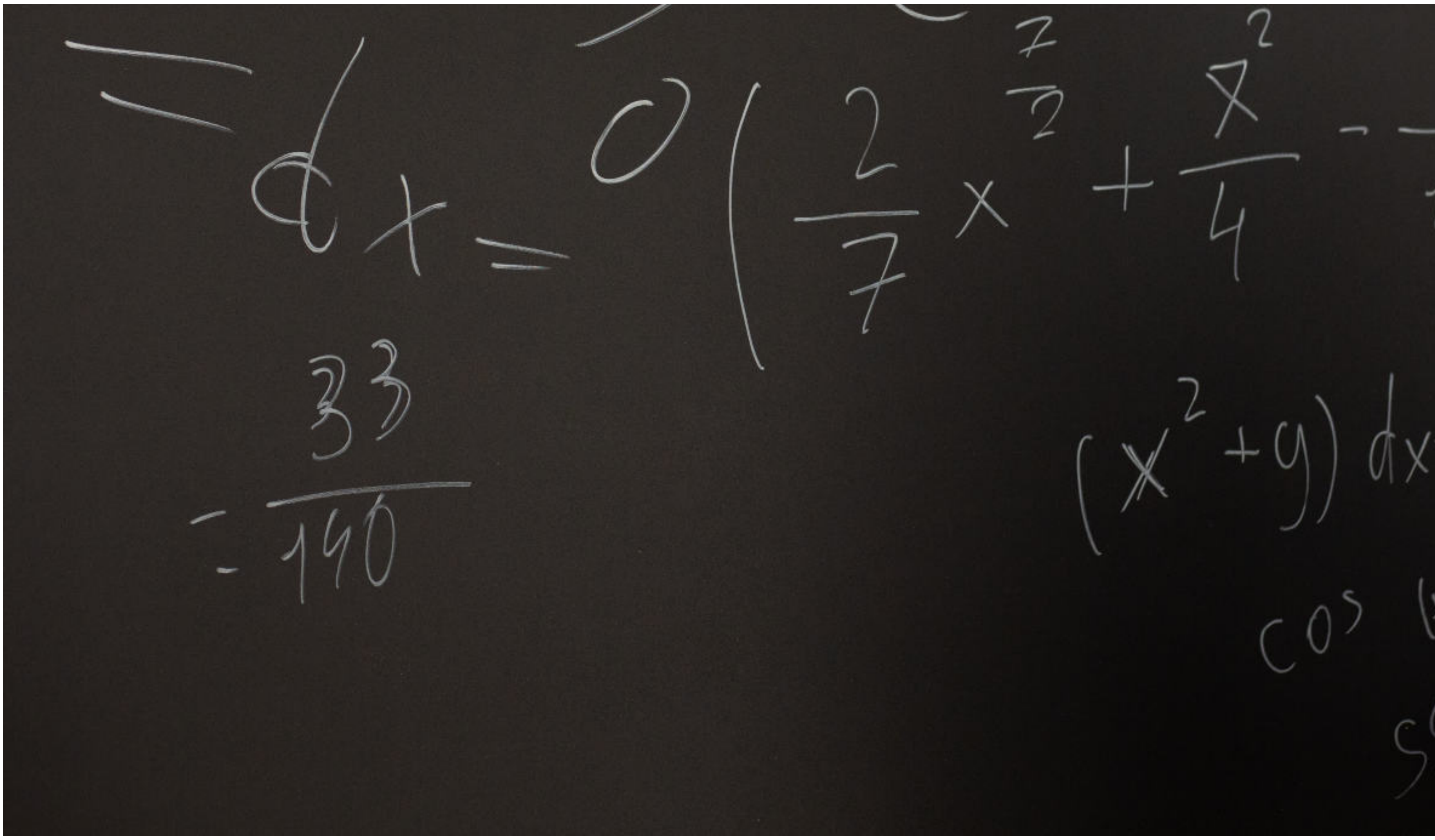
A Primer: Accessing services in Kubernetes

This article was fetched from an rss feed(04/02/2022)



Whenever I work on a local or remote Kubernetes cluster, I tend to want to connect to my application to send it a HTTP request or something similar.

> There are a number of options for doing this and if (like me), you work with Kubernetes on a daily basis, it's important to know the most efficient option and the best fit for our own needs.

Let's say that we're trying to access a Grafana dashboard? We could use any application here that receives HTTP traffic. TCP traffic is similar, but one option I'll show you (Ingress) won't work for TCP traffic.

I will use arkade to fetch the CLIs that we are going to need for the examples, but you are welcome to do things the "long way" if you prefer that. (Searching for a README, following lots of links, looking for the latest stable version, downloading it, unzipping it etc)

> arkade is an open-source Kubernetes marketplace for helm charts, apps and DevOps CLIs

```
arkade get kind
arkade get kubectl@v1.22.1

kind create cluster

# Install Grafana via helm with sane defaults
arkade install grafana
```

This creates a Service of type ClusterIP, which we cannot access from outside of the cluster:

```
kubectl get svc -n grafana
NAME      TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
grafana   ClusterIP   10.96.19.104   <none>        80/TCP    9d
```

Let's take a look at LoadBalancers, NodePorts, port-forwarding and Ingress. As a spoiler: port-forwarding tends to be the lowest common denominator which will work on any cloud or local distribution mostly seamlessly. For production, you will almost always deploy an Ingress Controller and expose port 80 and 443 via a managed LoadBalancer.

> I'm not going to cover any specific feature of a local distribution such as KinD's extra port mappings or Minikube add-ons.
>
> Let's keep this focused on Kubernetes API objects and approaches that work everywhere, without implying that one local or remote distribution is better than another.

On a side-note, did you know that minikube can now run in Docker instead of using VirtualBox? For me, that puts it on par with KinD and K3d. Both minikube and KinD both work really well on Apple Silicon with Docker Desktop. Another bonus for me.

LoadBalancer

A TCP load-balancer is offered by most managed clouds, you can allocate a port such as 8080, 443, etc and have a piece of infrastructure created to allow access to your Service.

For Grafana, the port of the internal service is `80`, so we'd create a LoadBalancer either through YAML or through `kubectl expose`.

```
kubectl expose -n grafana deploy/grafana \
  --target-port=80 \
  --port=8080 \
  --name grafana-external \
  --type LoadBalancer

service/grafana-external exposed
```

This will create an LB for port 8080 and we'll have to pay AWS or GKE ~ 15-20 USD / mo for this.

Here's the equivalent YAML via `kubectl get -n grafana service/grafana-external -o yaml`

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: grafana
  name: grafana-external
  namespace: grafana
spec:
  ports:
  - nodePort: 30180
    port: 8080
    protocol: TCP
    targetPort: 80
  selector:
    app.kubernetes.io/instance: grafana
    app.kubernetes.io/name: grafana
  type: LoadBalancer
status:
  loadBalancer: {}
```

The `status` field will be populated with a public IP address when using a managed K8s engine.

However, if we're running locally, we will never get an IP address populated and our LoadBalancer will be useless.

There's two solutions here:

1. Install the inlets-operator, which will create a virtual machine on a public cloud and connect us using an inlets Pro tunnel. The advantage here is that the LB will be just like a cloud LB, and accessible publicly. The inlets tunnel can work on any kind of network since it uses an outgoing websocket, even on a captive WiFi portals, VPNs, and behind HTTP proxies. See also: `arkade install inlets-operator`
2. Option two is to use something like the MetalLB project (see `arkade install metallb`). This project can allocate a local IP address to the LB and advertise it on our local network range with ARP. At that point we can access it locally on our own LAN at home, but not anywhere else. If we're on the road and plug our laptop into another network with a different network range, or connect to the VPN or go to the office, then we will probably run into issues.

Unfortunately, exposing a HTTP service over a LoadBalancer is less than ideal because it adds no form of encryption such as TLS. Our traffic is in plaintext. In the final section I will link you to a tutorial to combine Ingress with a LoadBalancer for secure HTTPS traffic on port 443.

NodePorts

A NodePort is closely related to a LoadBalancer, in fact if you look at the YAML from our previous example, you'll note that LoadBalancers require a node-port to be allocated to operate.

A NodePort is a port that is allocated in a high port range such as 30080. Any machine in your cluster that receives traffic on port `30080` will forward it to the corresponding service.

Pros: works on most public clusters, no need to pay for an LB.
Cons: doesn't always work on Docker Desktop due to the way networking is configured. Doesn't work well with KinD. Port number looks suspicious.

```
kubectl expose -n grafana deploy/grafana \
  --target-port=80 \
  --port=8080 \
  --name grafana-external \
  --type NodePort

service/grafana-external exposed
```

The NodePort will be allocated randomly. My service got: `32181`.

Here's the equivalent YAML via `kubectl get -n grafana service/grafana-external -o yaml`

If you write your own YAML file, you can specify a port for the NodePort, but make sure it doesn't clash with others that you've created.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: grafana
  name: grafana-external
  namespace: grafana
spec:
  ports:
  - nodePort: 30080
    port: 8080
    protocol: TCP
    targetPort: 80
  selector:
    app.kubernetes.io/instance: grafana
    app.kubernetes.io/name: grafana
  type: NodePort
status:
  loadBalancer: {}
```

Note that we are still working with plaintext HTTP connections, with no encryption.

Port-forwarding

Port-forwarding is my preferred option for accessing any HTTP or TCP traffic within a local or remote cluster. Why?

Pros: works with remote and local clusters, is encrypted by default using TLS. No special steps required
Cons: clunky user-experience, often disconnects, always needs to be restarted when a pod or service is restarted. Will not load-balance between replicas of a deployment, binds only to one pod.

So there are some significant cons with this approach, namely that if you are redeploying or restarting a service which you forwarded, then you have to kill the port-forward command and start it over again. It's a very manual process.

```
kubectl port-forward \
 -n grafana \
 svc/grafana 8080:80
```

You must also pay attention to ports, and avoid clashes:

```
Unable to listen on port 8080: Listeners failed to create with the following errors: [unable to create listener: Error listen tcp4 127.0.0.1:8080: bind
error: unable to listen on any of the requested ports: [{8080 3000}]
```

So let's change the port to 3001 instead:

```
kubectl port-forward \
 -n grafana \
 svc/grafana 3001:80

Forwarding from 127.0.0.1:3001 -> 3000
Forwarding from [::1]:3001 -> 3000
```

Now access the service via `http://127.0.0.1:3001`

But what if you want to access this port-forwarded service from another machine on your network? By default it's only bound to localhost for security reasons.

```
curl -i http://192.168.0.33:3001
curl: (7) Failed to connect to 192.168.0.33 port 3001: Connection refused


kubectl port-forward \
 -n grafana \
 svc/grafana 3001:80 \
 --address 0.0.0.0

Forwarding from 0.0.0.0:3001 -> 3000
```

Now try again:

```
curl -i http://192.168.0.33:3001/login

HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: text/html; charset=UTF-8
Expires: -1
Pragma: no-cache
X-Frame-Options: deny
Date: Fri, 04 Feb 2022 11:20:37 GMT
Transfer-Encoding: chunked

<!DOCTYPE html>
<html lang="en">
```

Exercise caution with `--address` - it will allow any machine on your network to access your port-forwarded service.

But there's still an issue here. Whenever we restart grafana because of a config change, look what happens:

```
kubectl rollout restart -n grafana deploy/grafana
deployment.apps/grafana restarted

kubectl port-forward -n grafana svc/grafana 3001:80 --address 0.0.0.0
Handling connection for 3001

E0204 11:21:49.883621 2977577 portforward.go:400] an error occurred forwarding 3001 -> 3000: error forwarding port 3000 to pod 5ffadde834d4c96f1d0f634e2
```

Now the only solution is find the tab running `kubectl`, kill the process with Control + C and then start over with the command again.

When you're iterating on a service over and over, as I often do, this grows tedious very quickly. And when you're port-forwarding 3 different services like OpenFaaS, Prometheus and Grafana it's that pain x3.

what if we had three replicas of the OpenFaaS gateway, and then port-forwarded that with kubectl?

```
arkade install openfaas
kubectl scale -n openfaas deploy/gateway --replicas=3
kubectl port-forward -n openfaas svc/gateway 8080:8080
```

Unfortunately, if you have three replicas of a Pod or Deployment, `kubectl port-forward` will not load-balance between them either. So is there a smarter option?

Smarter port-forwarding

With inlets, you can set up your local machine (laptop) as a tunnel server, then deploy a Pod to the cluster running a client. This is the opposite way that inlets is usually used.

> inlets was originally created in 2019 to bring tunnels to containers and Kubernetes, so you could expose local services on the Internet from any network.

The K8s cluster will run the inlets client which is a reverse proxy. The server portion on my local machine will send requests into the cluster and then they will be send onto the right pod.

This port-forwarding with inlets only uses up one port on your host, and will allow you to access a number of services at once. If you restart pods, that's fine, there's nothing for you to do unlike with `kubectl port-forward`.

Inlets will also load-balance between any pods or services within your cluster, where as `kubectl port-forward` only binds to a single port, which is why it has to be restarted if that pod gets terminated.

inlets.yaml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inlets-client
spec:
  replicas: 1
  selector:
    matchLabels:
      app: inlets-client
  template:
    metadata:
      labels:
        app: inlets-client
    spec:
      containers:
      - name: inlets-client
        image: ghcr.io/inlets/inlets-pro:0.9.3
        imagePullPolicy: IfNotPresent
        command: ["inlets-pro"]
        args:
        - "http"
        - "client"
        - "--url=wss://192.168.0.33:8123"
        - "--upstream=prometheus.svc.local=http://prometheus.openfaas:9090"
        - "--upstream=gateway.svc.local=http://gateway.openfaas:8080"
        - "--upstream=grafana.svc.local=http://grafana.grafana:80"
        - "--token=TOKEN_HERE"
        - "--license=LICENSE_HERE"
---
```

Set the `--token` and `--license` then run: `kubectl apply -f inlets.yaml`. Change the IP from `192.168.0.33` to your local machine's IP.

Check the logs, it's detected our upstream URLs and is ready to go. You can see where each hostname will go to in the cluster including the namespace.

```
kubectl logs deploy/inlets-client

2022/02/04 10:53:52 Licensed to: alex <alex@openfaas.com>, expires: 37 day(s)
2022/02/04 10:53:52 Upstream: prometheus.svc.local => http://prometheus.openfaas:9090
2022/02/04 10:53:52 Upstream: gateway.svc.local => http://gateway.openfaas:8080
2022/02/04 10:53:52 Upstream: grafana.svc.local => http://grafana.grafana:80
time="2022/02/04 10:53:52" level=info msg="Connecting to proxy" url="wss://192.168.0.33:8123/connect"
time="2022/02/04 10:53:52" level=info msg="Connection established" client_id=65e97620ff9b4d2d8fba29e16ee91468
```

Now we run the server on our own host. It's set up to use encryption with TLS, so I've specified the IP of my machine `192.168.0.33`.

```
inlets-pro http server \
  --auto-tls \
  --auto-tls-san 192.168.0.33 \
  --token TOKEN_HERE \
  --port 8000
```

The `--port 8000` value sets port 8000 as where we will connect to access any of the forwarded services.

Now, edit `/etc/hosts` and add in the three virtual hosts we specified in the inlets-client YAML file and save it.

```
127.0.0.1       prometheus.svc.local
127.0.0.1       gateway.svc.local
127.0.0.1       grafana.svc.local
```

I can now access all three of the services from my machine:

```
curl http://prometheus.svc.local:8000
curl http://gateway.svc.local:8000
curl http://grafana.svc.local:8000
```

If you have more services to add, just edit the above steps and redeploy the inlets client. The tunnel server doesn't need to be restarted.

Note that none of this traffic is going over the Internet, and it is ideal for a local development environment with WSL, Docker Desktop and VMs.

Ingress

Kubernetes Ingress is the last option we should cover.

Ingress isn't actually a way to access services directly, it's more of a way to multiplex connections to different back-end services.

Think of upstreams in Nginx, or VirtualHosts in Apache HTTP Server.

To make use of Ingress, you need to install an IngressController, of which there are dozens.

The most popular (in my opinion) are:

- ingress-nginx
- Traefik

There are many many others.

Once an Ingress Controller is deployed, you then have a chicken and egg problem again. It has a TCP port 80 and 443, and you have to expose that somehow.

Your options are the same as above:

To access it publicly: a managed cloud LoadBalancer or inlets-operator

To access it locally: MetalLB, NodePorts or port-forwarding.

Remember that NodePorts don't give out good ports like 80 and 443, do you really wan to go to `https://example.com:30443`? I didn't think so.

> With Kubernetes, so many things are possible, but not advisable. You can actually change security settings so that you can bind NodePorts to port 80 and 443. Do not do this.

You can install ingress-nginx with arkade:

```
arkade install ingress-nginx
```

You'll then see its service in the default namespace:

```
kubectl get svc ingress-nginx-controller
NAME                     TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx-controller LoadBalancer   10.96.213.226   <pending>     80:30208/TCP,443:31703/TCP   3d20h
```

This is type of LoadBalancer, because it's expected.

I'm going to install the inlets-operator and have it provision IPs for me on DigitalOcean VMs.

```
arkade install inlets-operator \
  --provider digitalocean \
  --region lon1 \
  --token-file ~/do-token.txt
```

I got the API token `do-token.txt` from DigitalOcean's web portal and created a token with Read/Write scope.

Now, almost as soon as that command was typed in, I got a public IP:

```
kubectl get svc -n default -w
NAME                                 TYPE           CLUSTER-IP      EXTERNAL-IP                    PORT(S)                         AGE
ingress-nginx-controller             LoadBalancer   10.96.213.226   178.128.36.66   80:30208/TCP,443:31703/TCP   3d20h
```

```
curl -s http://178.128.36.66

Not found
```

Let's create a basic Ingress record that will always direct us to Grafana.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: grafana
  namespace: grafana
spec:
  ingressClassName: nginx
  rules:
  - http:
      paths:
      - backend:
          service:
            name: grafana
            port:
              number: 80
        path: /
        pathType: Prefix
status: {}
```

This ingress record matches any traffic and then sends to to `grafana.grafana:80` within our cluster.

Typically, when we're using Ingress it's to get some benefit of a reverse proxy like being able:

- apply rate limits
- authenticate and gate requests
- load-balance between services
- multiplex a number of domains under one IP or LoadBalancer
- encrypt traffic with a TLS certificate

I've just given you a brief example of how to use Ingress, but you can follow this tutorial to see how to use multiple domains and how to get TLS records:

- Tutorial: Expose a local IngressController with the inlets-operator

What about a "Service Mesh"? Linkerd advocates for using ingress-nginx or similar to get incoming network access. Istio ships its own Gateway called an IstioGateway which can replace Ingress on Kubernetes. For an example of Istio with OpenFaaS see: Learn how Istio can provide a service mesh for your functions (adapt as necessary for your workloads).

Wrapping up

We looked at LoadBalancers, NodePorts, kubectl and inlets port-forwarding and Ingress. This is not an extensive guide, but I hope you have a better handle on what's available to you and will go off to learn more and experiment.

If, like me you do a lot of application development, you may find inlets useful and more versatile vs `kubectl port-forward`. We showed how to multiplex 3 different HTTP services from our cluster, but inlets also supports TCP forwarding and a developer at UK Gov wrote to me to explain how he used this to debug NATS from a staging environment.

You may also like: Fixing the Developer Experience of Kubernetes Port Forwarding

So what is my preferred setup?

For a production cluster, serving websites and APIs over HTTP - the "de facto" setup is a managed cloud LoadBalancer to expose TCP port 80 and 443 from your Ingress Controller.

As I explained in the introduction, our focus is on local development techniques that work for every kind of local Kubernetes cluster. We are not trying to tie our destinies to KinD or Minikube or Microk8s etc.

I will often set up the inlets-operator, cert-manager and ingress-nginx on my local environments so that I can get full TLS certs and custom domains during local development. This mirrors the recommended configuration for a production setup and also works with Istio.

I also use port-forwarding (`kubectl port-forward`) a lot, as I mentioned in the introduction - it's the lowest common denominator and, if you have access to kubectl, it will work everywhere.

I use either a Linux desktop with Docker CE installed and KinD for Kubernetes, or my Apple M1 MacBook Air with Docker Desktop and KinD. Arkade is smart enough to download the right version of KinD, kubectl, inlets-pro and etc by looking at the output of `uname -a`.

More recently, I've found myself needing access to Grafana, OpenFaaS and Prometheus all at once, and my example with an inlets client running as a Pod gave me a huge productivity boost over kubectl. Your mileage may vary. You can try inlets on a monthly subscription with no long-term commitment. There are other tools available that specialise in port-forwarding for local development, but I am biased since I created inlets to solve problems that I was facing which these do not.

Getting more in-depth experience

If you want to gain experience with Kubernetes objects and APIs, why not try the course I wrote for the LinuxFoundation? It covers all the basics whilst also introducing you to K3s:

Introduction to Kubernetes on Edge with K3s

I also have another course commissioned by the Cloud Native Computing Foundation (CNCF) that covers Serverless on Kubernetes with OpenFaaS being used for many of the more detailed examples:

Introduction to Serverless on Kubernetes

Premium materials

I also have my own premium learning materials available in eBook, code sample and video format. Feel free to browse the table of contents and see what others are saying.

Learn use-cases for functions and how to build our own with Node.js: Serverless for Everyone Else

Start contributing to Cloud Native projects like Kubernetes and OpenFaaS, or write your own code in Go with my book - Everyday Go