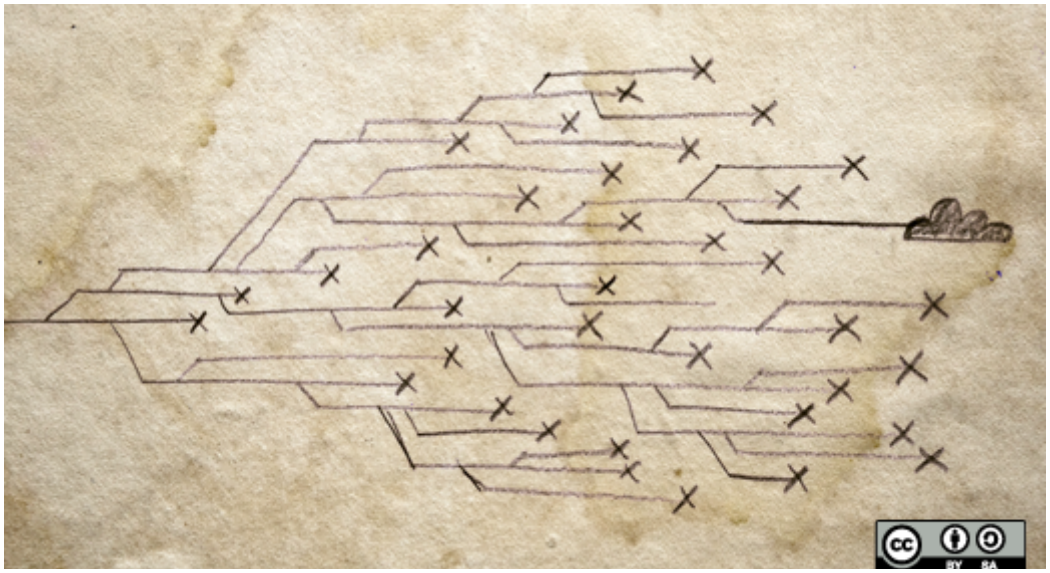
Why I use Ingress Controllers to expose Kubernetes services

Kubernetes ingress controllers will make or break your cloud architecture.

03 Aug 2020 | Kevin Crawley (/users/kevin-crawley) | 45



The meteoric rise of containerization and microservices has been necessary to meet the growing demand for applications, but getting it right means overcoming some critical network orchestration challenges. Out of the complexities that developers of cloud-native applications face, strategically utilizing Kubernetes (https://opensource.com/resources/what-is-kubernetes) ingress controllers is among the most difficult components to understand—and among the most important.

Before diving into ingress controllers, you need to understand why networking is so important to developer workflows.

It is common for development teams to create backend API services to enable connectivity for external applications and users. In early development phases, teams often use implementations of container environments on local development machines, which more simply rely on direct container invocations through Docker Compose (https://docs.docker.com/compose/) or similar local orchestrators for access.

However, when the time comes to shift to a shared development or staging environment and match the configuration that will be used in production, these direct-access stopgaps are no longer sufficient. The access patterns often assume trusted access, which can't be assumed in production, or they rely on static values that are likely to change in a cloud infrastructure.

The aforementioned stopgap is often handled through port mapping—literally exposing applications to the local machine through an arbitrarily assigned port. But port mapping doesn’t translate at all to applications running in a Kubernetes environment, as there frequently are multiple replicas running and, oftentimes, there will be clusters running the multiple versions of the software. While it’s possible to use Kubernetes pods and deployment resources to bring automation into container lifecycle management when making this transition, these solutions don’t enable application access.

Fortunately, Kubernetes features a dedicated resource abstraction to fulfill this

Accessing applications as Kubernetes services

Containers deployed within Kubernetes pods receive designated Internet protocol (IP) addresses, but various lifecycle operations can alter these addresses. This makes it particularly challenging to ensure that other components can locate (and connect with) containers. To address this—no pun intended—Kubernetes offers a defined service resource that can automatically manage this connectivity.

Developers can define Kubernetes services associated with pods and specify the service names that they can be accessed by. Depending on the service type used when configuring these services, connections can be established and implemented in a variety of ways. Often-used Kubernetes services (https://kubernetes.io/docs/concepts/services-networking/service/) types include ClusterIP, NodePort, and LoadBalancer.

ClusterIP

If the service resource definition doesn't have a service type defined, ClusterIP is selected by default. This service type prompts Kubernetes to create a virtual cluster IP address for pod connectivity. However, the virtual cluster IP address can only be routed *within* the cluster. For this reason, ClusterIP services are most appropriate for use cases where they enable mutual exposure of internal-only application endpoints.

NodePort

- eBook: Getting started with Kubernetes (https://opensource.com/kubernetes-dump-truck?intcmp=7016000000127cYAAQ)
- An introduction to enterprise Kubernetes (https://www.redhat.com/en/resources/managing-containers-kubernetes-openshift-technology-detail?intcmp=7016000000127cYAAQ)
- How to explain Kubernetes in plain terms (https://enterprisersproject.com/article/2017/10/how-explain-kubernetes-plain-english?intcmp=7016000000127cYAAQ)
- Latest Kubernetes articles (https://opensource.com/tags/kubernetes?intcmp=7016000000127cYAAQ)

The simplest approach to enabling external access to services is using the NodePort (https://kubernetes.io/docs/concepts/services-networking/service/#nodeport) service type. NodePort opens a specified port on every node in the cluster. An underlying ClusterIP is used to route clients that connect to an exposed node port.

The NodePort service type offers advantages and disadvantages. On the positive side, NodePort effectively provides communication for applications outside of the cluster. On the downside, services must use a limited range of ports (by default, in the 30000 to 32767 range), and just a single port can be mapped to a single service. Also of note: if the host's IP address or virtual machine supporting the node that clients connect through changes, that information must be updated.

LoadBalancer

The LoadBalancer (https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer) service type is typically leveraged in cloud environments, and it automates the provisioning of load balancers external to the Kubernetes cluster. Using LoadBalancer cleanly sidesteps the issues of temporary IP addresses while also fully supporting network access for external applications. But user beware: this service type can escalate cloud provider costs by adding an additional hop between the cloud-based load balancer and ingress controller

ClusterIP, NodePort, and LoadBalancer offer useful methods for establishing external access to services that are appropriate under certain circumstances. That said, the limitations of each of these Kubernetes networking options understandably make many application developers look for an alternative.

The ingress resource abstraction

An ingress resource abstraction native to Kubernetes exposes HTTP and HTTPS endpoints and enables rules to be defined that control traffic routing. This technique is ideal for application developers and DevOps teams because the ingress resource makes it possible to expose multiple services using a singular external endpoint, a load balancer, or both at once. Taking this approach, teams can enact host, prefix, and other rules to route traffic to defined service resources however they prefer.

The ingress resource abstraction enhances service resource capabilities to enable external access with more flexibility. The job isn't done yet, though: Defining an ingress resource alone only sends a request for networking configuration. Another component is needed to complete the work of allowing access to services from outside Kubernetes.

Ingress controllers

When it comes to acting on requests for ingress resources, ingress controllers address inbound requests and produce the necessary routing specifications in line with the specific technology at hand. In common use cases, various ingress controllers are installed within a Kubernetes cluster, where they can be selected and deployed to address each request as appropriate.

A few popular examples of ingress controllers include:

- **Traefik** (https://docs.traefik.io/)**:** This is known as a simple and reliable open source Kubernetes ingress controller designed for connecting Kubernetes with external applications

This commonly used ingress controller leverages AWS Application Load Balancers to handle ingress resource requests.

- **Nginx** (https://www.nginx.com/products/nginx/kubernetes-ingress-controller/): This controller uses open source Nginx to implement ingress resources.

Selecting an ingress controller

Developers don't risk missing out when selecting ingress controllers: it's possible to deploy multiple options to Kubernetes and use whichever is most beneficial for each task. That said, there are key factors for a team to consider in choosing ingress controllers. Tools that support traffic splitting based on customer weight definitions offer developers an increased range of options. The ability to utilize flexible route definitions, name and path-based routing, prioritized routes, and custom resource definitions (CRDs) are also worth looking for. From a security perspective, support for Let's Encrypt (https://letsencrypt.org/) and automated certificate management are good to have.

But above all, developer teams should identify and implement ingress controllers that best fit their use cases. Having a plan and deploying ingress controllers correctly will make it far simpler to wield the full potential of Kubernetes in application development—and to do so without committing unnecessary time and resources to networking.