Application Networking in Kubernetes

Ingress, service mesh, Gateway API

- Kubernetes network model in a nutshell
- In-depth analysis of available solutions for application networking in Kubernetes: Ingress, service mesh, Gateway API
- Sample use cases and configurations



codilime

Executive summary

Publication purpose

This publication presents **solutions** for **application networking** in the **Kubernetes** environment, including the differences between them and example use cases.

Aspects featured in this publication

- An overview of the network model used by the Kubernetes platform
- **Ingress** – a relatively simple solution that performs L7 routing. Ingress allows handling of so-called north-south traffic, i.e. the traffic entering a cluster from outside which is then routed to various workloads deployed in the cluster.
- **service mesh** – a system enabling fine-grained control of the internal traffic between workloads running in a cluster, i.e. so-called east-west traffic.
- **Gateway API** – a solution that offers a wide set of advanced functionalities (in a standard way) that are not supported by the native Ingress (thus it can be considered as its successor).

Kubernetes as a platform is becoming more and more ubiquitous. The solutions that operate in this ecosystem are subject to constant evolution and development. This also includes the subject of application networking. Solutions architects, application developers and product owners should definitely be up to date with this area.

Table of contents

Executive summary 2

Kubernetes networking in a nutshell 4

Kubernetes Ingress 4

Service mesh 7

Ingress vs. service mesh – what is the difference? 11

Kubernetes Gateway API 14

Ingress vs. Gateway API – example use case 16

Summary 25

About CodiLime 27

Kubernetes networking in a nutshell

The **network model** in Kubernetes has its own specifics. The fundamental issue here is how containers (Pods) are attached to the network. This includes specific aspects such as:

- How to connect/remove a container to/from a network in the cluster?
- What should be configured inside the container (an interface, IP address, a default route, other routes, other parameters like MAC address, MTU, bandwidth, etc.)?
- What should be configured on the host (Kubernetes node) level?
- What technologies can be used for that purpose?

In Kubernetes, these are done based on the CNI specification, which provides a standard way to perform the operations mentioned above. In practice, CNI plugins (sometimes also called network plugins) are responsible for this. There are many different implementations of such plugins that are compliant with the specification. For example, the CNI project provides a group of so-called reference plugins. However, in production-grade clusters third-party plugins are usually installed. Some of them are relatively simple, while others are more advanced in terms of the functionality and/or technologies they are based on. Examples of third-party plugins are Calico, Cilium, Flannel, kube-ovn, etc.

After the Pods have been attached to the network in the cluster, the question is how should they communicate with each other and with the outside world? To address this issue, Kubernetes introduces **Services**. There are several different types of Services available that you can take advantage of. Each type has a slightly different purpose. While Services are actually a kind of abstraction only, there is also **kube-proxy**, a Kubernetes component that implements the logic behind Services. Another solution Kubernetes offers out of the box is an internal DNS implementation, called **CoreDNS**. Thanks to this, resources deployed in a cluster can use not only IP addresses but also domain names when they talk to each other.

All the functionality described so far is available for whatever type of user workload is running in the Kubernetes cluster. The vast majority of these mechanisms operate at the L2-L4 level. However, this is sometimes not enough for some **application workloads** running in a cluster. Sometimes, for the best performance, support for **L4+** network mechanisms is also required. This publication presents three solutions that help handle applications' specific needs in terms of networking. These are **Ingress**, **service meshes** and **Gateway API**.

Kubernetes Ingress

What is Ingress?

Ingress is a Kubernetes API object that manages external access to multiple resources running in a cluster. Usually, these resources are Kubernetes Services. Underneath, Ingress is a kind of proxy, typically for HTTP or HTTPS traffic, so it works at L7. More specifically, it implements **L7 routing**, which is also sometimes called **HTTP routing** or **path-based routing**. In short, Ingress exposes HTTP routes from outside the cluster to Services in the cluster. For example, if requests containing different endpoints, e.g. `example.com/api/v1`, `example.com/blog`, `example.com/stats` are received by the Ingress controller they can be routed to three different Services deployed in the cluster.

To better understand this, consider the situation shown in Fig. 1. Let us assume that an online store has been deployed in Kubernetes. It allows you to buy three different types of products: books, videos and games. Also assume that the process of purchasing a given type of product is handled by a dedicated application, launched as a Kubernetes Deployment and represented by the given Pods. Each of these applications also has its own Service configured.

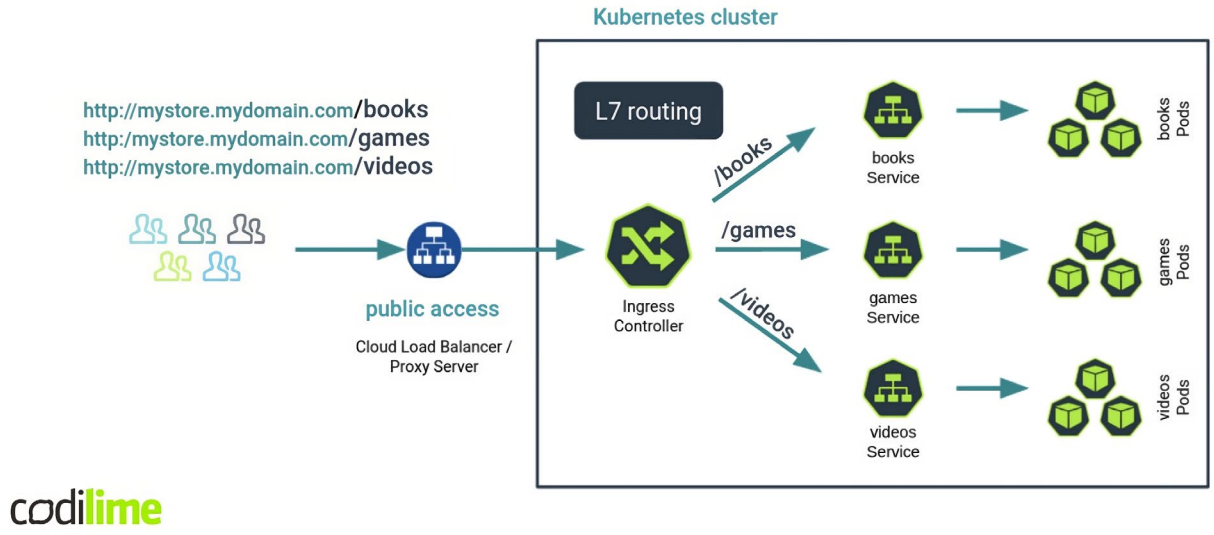


Fig. 1 Kubernetes Ingress – example scenario

The Ingress object configuration is presented below. This configuration ensures Ingress accepts requests to `mystore.mydomain.com` and redirects them to the Services corresponding to the given applications, based on arbitrarily defined **prefixes**.

```yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: my-ingress
spec:
 rules:
 - host: mystore.mydomain.com
   http:
     paths:
     - pathType: Prefix
       path: /books
       backend:
         service:
           name: books
           port:
             number: 80
     - pathType: Prefix
       path: /games
       backend:
         service:
           name: games
           port:
             number: 80
     - pathType: Prefix
       path: /videos
       backend:
         service:
           name: videos
           port:
             number: 80
```

ingress.yaml

The above configuration (at the Ingress level) is responsible only for HTTP routing within the cluster. It does not make individual resources deployed in the cluster automatically available from the Internet. For this, additional network mechanisms are needed. But we will not go into detail about these since they may differ depending on the environment in which the Kubernetes cluster is running (cluster in the cloud vs. on-premise cluster).

Implementations

It is important to understand that to make Ingress work an underlying mechanism executing Ingress logic is needed. This is called an **Ingress controller**. Creating a "dry" Ingress object without having any Ingress controller running in the cluster has no practical effect. Kubernetes does not offer any native Ingress controller implementation by itself. Instead, you are free to choose one of the many Ingress controller solutions available, such as:

- Nginx Ingress Controller,
- Kong,
- Contour,
- Gloo Edge,
- Voyager

Implementations are also offered by cloud Kubernetes providers (AWS, GCP, Azure) and are well-integrated with their cloud services, etc. Usually these are based on one of the available reverse proxy engines, such as Envoy, Nginx, or HAProxy, etc.

Service mesh

What is a service mesh?

Over the past few years, we have seen a shift away from approaches based on monolithic code when designing software applications. Instead, modern design is based on **microservices architecture**. At the end of the day, this is about delivering basically the same business logic, not in the form of a large monolith but as a collection of loosely coupled and independently deployable components. These are called **microservices** or just **services**, not to be confused with Kubernetes Services.

Why do this? Well, it has undoubted advantages. For example, microservices can have their own development cycle, can be developed using different programming languages and frameworks, and can be owned and managed by different teams. Such a decomposition into various functional modules allows you to optimize the delivery process of the entire application. Extending the application with new functionalities and updating or replacing individual functional pieces with new ones becomes much easier. What is more, microservices architecture is by nature perfectly suited to containerization-based deployments (e.g. in such environments as Kubernetes) which is certainly a very powerful feature, especially today.

However, it is worth noting that this approach also brings considerable challenges. If you want an application based on microservices to work reliably, efficiently and meet security requirements

(this is kind of a “must” when building modern, scalable, server-side applications) each individual microservice must have appropriate mechanisms that ensure this.

What is important here is that all aspects relating to proper communication and interaction between microservices are complex. Theoretically, they could be handled by the microservices’ development teams through appropriate implementations within the services themselves, but this is not an optimal approach. Firstly, because these tasks are rather time-consuming, they require some effort and are therefore prone to errors. Secondly, developers should focus on implementing the actual business logic of the given microservice.

This is where the **service mesh** concept comes in handy. A service mesh is a kind of “network” that connects all the microservices within a given application. In fact, it is extra software that acts as an intermediary layer between services and provides functions such as service discovery, service-to-service authentication, load balancing and traffic management, monitoring, resilience, etc.

System components

The logical architecture of a service mesh is quite simple because it only contains two components: a **control plane** and a **distributed data plane** (see Fig. 2 below). The data plane consists of proxies installed next to each service instance. They basically act as L7 proxies (working in both “forward” and “reverse” mode) and handle calls to and from the services. When a particular service mesh implementation (such as Istio or Linkerd, for example) is deployed in a Kubernetes cluster the proxies are **sidecar** containers that run in the same Pods as the service containers. The control plane provides the core functionality of the service mesh system. In short, it configures and coordinates the behavior of the proxies.

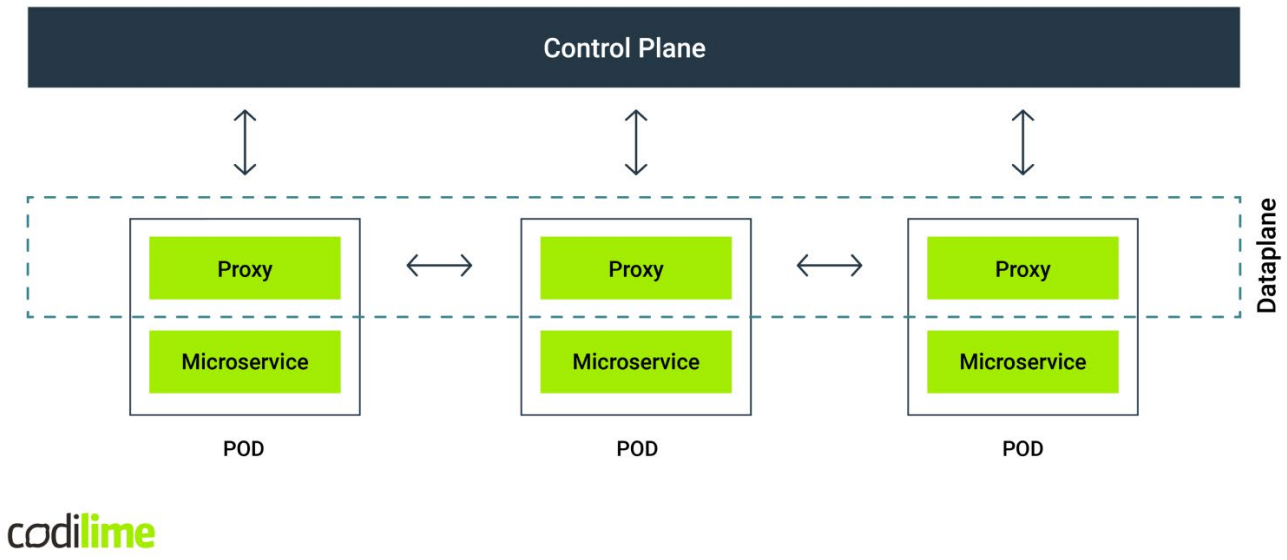


Fig. 2 Service mesh architecture overview

What features does a service mesh offer?

So what can service mesh do for you? Well, depending on the specific implementation you want to use, the feature set may differ slightly. However, the key functions are laid out below, broken down into categories.

- **The first category encompasses functions related to routing and traffic management.**
  In short, it is about proxying and performing **sophisticated load balancing** (at L7) for various types of protocols, e.g. HTTP/1.x, HTTP2 (including gRPC), WebSocket and other TCP-based protocols.

- **The second category includes reliability and resilience mechanisms.**
  One example is the so-called **circuit breaker**, allowing you to block incoming connections to the application when given conditions are met (e.g. the maximum number of concurrent connections has been reached or a failure threshold defined for some app has been exceeded). Another example here are **timeouts and retries** that prevent workloads running in the cluster from being overwhelmed with requests (e.g. when a failure has occurred and the given application is temporarily unavailable).

- **Security features are another category.**
  Here you can apply advanced mechanisms for **authentication** and **authorization** as well as **mutual TLS** between workloads.

- **The last category is the so-called observability functions.**
  In general, **observability** can be defined as the ability to measure the current state of a system or software environment by examining the different types of data it generates. It is based on three pillars: **monitoring** (a process of continuous gathering of various types of metrics – numerical data, most often related to the system performance), **logging** (a process of registering all kinds of events that happened within the system) and **tracing** (a process that helps to observe and analyze the entire life cycle of information flows going across the system). In the context of a service mesh, observability is about collecting various types of metrics, e.g. request volumes, success rates, latencies, etc. (monitoring), gathering the data needed to troubleshoot latency issues (tracing), and providing verbose logging capabilities. To make all these possible, a given service mesh solution can be integrated with different observability backends, such as Prometheus for monitoring, Zipkin or Jaeger for tracing, Fluentd for logging, etc.

As you can see, service meshes offer a rich set of functionalities. The following scenario shows a practical service mesh use case. Specifically, it is about sophisticated load balancing, which you can use to route traffic between workloads running in your cluster. First, imagine that you have built a quite complex application following the microservice architecture pattern (see Fig.3). The application has been launched in Kubernetes and the individual microservices work as Pods. One of the typical workflows within your application is that *frontend* Pods communicate with the *api* Pods. At some point you create a new version of the *api* module. You have already pre-tested it, and now you want to deploy it using the **canary rollout** strategy. This means that two versions of the *api* module are running simultaneously: a small part of traffic from the *frontend* Pods goes to the new version (to test it in a production environment), and the rest goes to the stable version. In consequence, you want to be able to distribute such traffic in a completely arbitrary manner, e.g 5% to 95%.

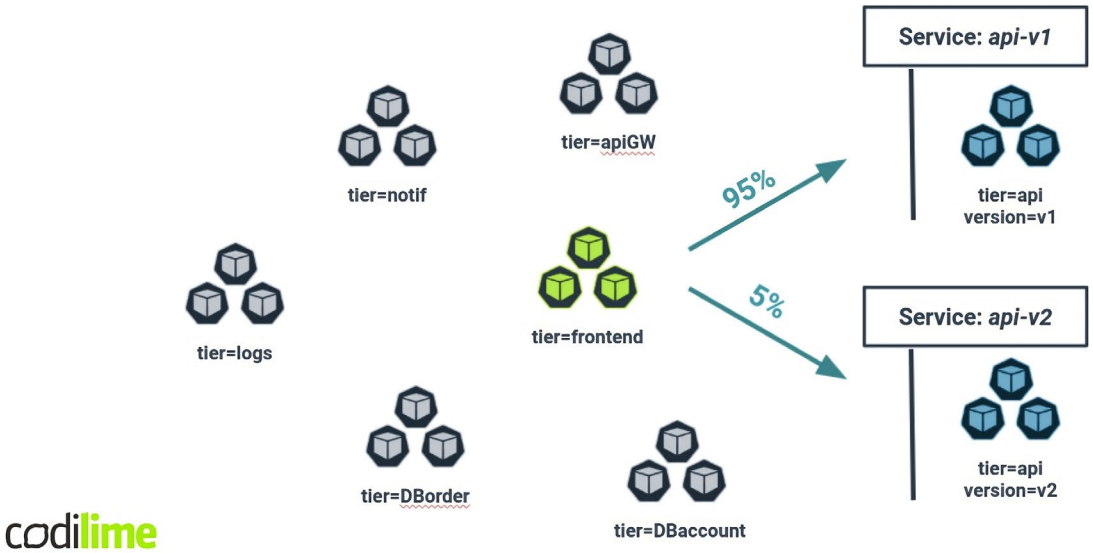


Fig. 3 Example of sophisticated load balancing (traffic split) offered by service mesh solutions

The below configuration allows you to accomplish this. It follows the SMI (Service Mesh Interface) specification. SMI is a CNCF project. As it covers the most common service mesh capabilities, its goal is to provide a common, portable set of service mesh APIs that can be used in a provider-agnostic manner.

```
---
apiVersion: split.smi-spec.io/v1alpha4
kind: TrafficSplit
metadata:
 name: api-ts
```

```
spec:
service: api-svc
 backends:
 - service: api-v1
   weight: 95
 - service: api-v2
   weight: 5
```

traffic-split.yaml

For simplicity, the presented configuration contains only the **TrafficSplit** object. In the considered scenario, this is a key element providing a load-balancing mechanism that allows for arbitrary traffic distribution. However, it should be remembered that the Services it refers to must also be configured. They must have selectors defined correctly, so that they point to the appropriate endpoints (the old and the new versions of the *api* module Pods).

As mentioned above, the example above refers to SMI. However, it is also worth knowing that the vast majority of currently available service mesh solutions have their own **CRDs** (custom resource definitions), usually much richer and more advanced. Although some solutions may support SMI, their CRDs are usually the preferred way to configure the functionality they offer.

Implementations

There are many different service mesh implementations available today. They differ both in terms of maturity and the number of supported functionalities. Example solutions are:

- Linkerd
- Istio
- Traefik Mesh
- Kuma
- Open Service Mesh
- Nginx Service Mesh
- Consul

You can get more information about each of them by following the corresponding links above.

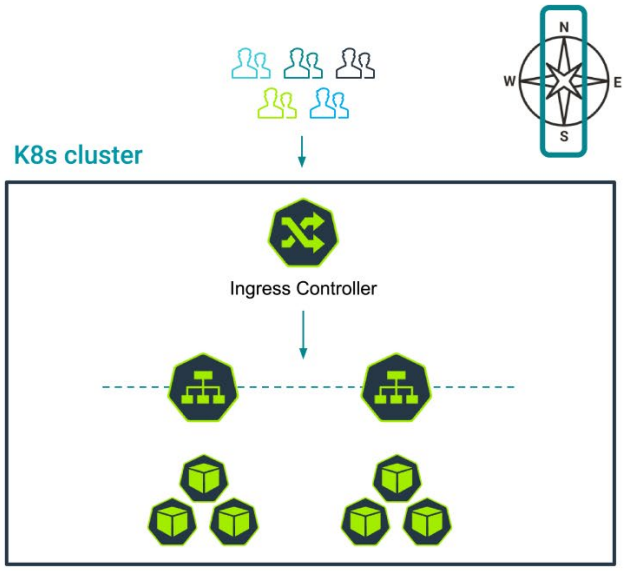
Ingress vs. service mesh – what is the difference?

If you are not familiar with Kubernetes Ingress and service mesh concepts or you have not had any prior experience with them, you could easily get the impression that they are basically similar solutions. They both process traffic on the L7... They both act as proxies... So where is the difference? Maybe it is about the set of offered functionalities?

Service mesh solutions usually offer a much richer feature set than the native Ingress does. However, you should be aware of one very important fact. Most of the Ingress controller solutions implement much more functionality than the Ingress API would normally offer. These features are exposed through the **CRDs** and **annotations**. Moreover, in many cases the extended functionality may be (more or less) similar to that offered by various service meshes. So the real difference lies elsewhere.

And here we come to the key point. It is very important to know how to correctly identify to which cases Ingress applies, and for which a service mesh is more suitable, because this may not always be clear.

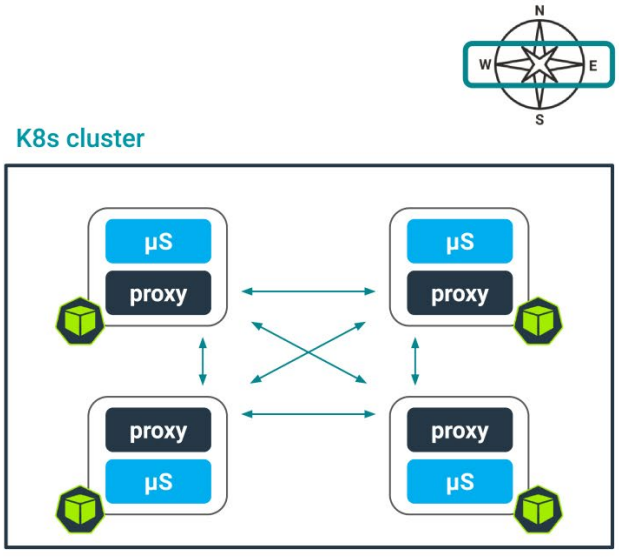
Let us go back to the scenario of an online store discussed earlier. It can be generalized to the case where you have deployed several applications in the same Kubernetes cluster. They are all completely independent workloads and have nothing to do with each other but each of them will handle HTTP(S) requests coming from outside the cluster. What you want to do is configure a common entry point for them that will allow external users to use the same domain name followed by a path specific to a given application when they connect to those applications using web browsers. This is an ideal use case for employing Kubernetes Ingress as it will allow you to handle the HTTP(S) traffic entering the cluster from outside and to route it to your applications appropriately. This is sometimes called **north-south** traffic (see Fig.4). If you assume external users can reach your applications from the Internet, the full picture requires further actions to be carried out, such as putting a cloud load balancer or on-premise reverse proxy server in front of the cluster, depending on where it is deployed, obtaining a public IP address, public domain name and its registration, and adding all the necessary configuration, etc.



codilime

Fig. 4 Kubernetes Ingress – handling north-south traffic

On the other hand, there might be cases where you need to control the **east-west** traffic (see Fig.5), meaning the traffic between workloads running in a cluster. A perfect example of this would be a scenario where you deploy in Kubernetes an application based on a microservices architecture that contains many components. If you want a deeper insight into the communication between services at the higher layers of the network stack and you need to control such traffic in a sophisticated way (e.g. using advanced load balancing and resilience mechanisms), then using a service mesh for this purpose will be a good idea.



codilime

Fig. 5 Service mesh in Kubernetes – handling east-west traffic

Of course, this is a very **general distinction**. There may also be use cases when you need to use Kubernetes Ingress and a service mesh at the same time. For example, if in the above-mentioned scenario (the application based on a microservices architecture) we add the assumption that multiple services are supposed to serve HTTP(S) requests coming from outside the cluster, then such traffic must be properly handled. Here, in addition to the service mesh, Ingress will also come in handy. By the way, it is worth knowing that some of the service mesh implementations offer such functionality out of the box, i.e. they allow you to control not only east-west but also north-south traffic. Then you do not need to onboard any additional Ingress controller into your cluster.

Kubernetes Gateway API

What is Gateway API?

As mentioned before, because Ingress itself is quite simple and therefore a bit limited in terms of the functionalities it can offer (the HTTP(S) routing it performs is based solely on host and/or path matching) to address more demanding scenarios, controllers from different vendors offer a whole range of **additional functionalities** beyond the native version of Ingress. They are provided in the form of CRDs and annotations. What is interesting is that they often appear to be quite similar to each other, though they differ in syntax.

Therefore, in 2019, discussions began on the evolution of Ingress and the need for a new API of this kind. One of the goals for such an API was to deliver most of the additional capabilities as part of the common core API standard. This resulted in the initiative which today is called Gateway API and which has been developed by Kubernetes' SIG-NETWORK (Special Interest Group – Network).

Gateway API provides a collection of resources among which three main object types can be indicated:

- **GatewayClass**
- **Gateway**
- **xRoutes** (e.g. HTTPRoute, TCPRoute, GRPCRoute, etc.)

Gateway is a logical representation of the underlying network infrastructure that processes the traffic (e.g. load balancer, proxy, etc.). Gateway, however, only offers a basic configuration. More granular configurations are defined within **xRoutes** that are attached to a given Gateway instance. Moreover, xRoutes can be configured separately by different users in the cluster. It is therefore a kind of decentralized configuration model. In short, it can be said that Gateway and xRoutes together perform the same role that Ingress performs as a single resource. **GatewayClass**, in turn, is something similar to the previous IngressClass. It indicates a controller that implements the functionality defined within the Gateway API.

Roles in the Gateway API

Such an approach may not seem entirely clear at first. It does become more justified, however, when you discover that the intention of the Gateway API designers was to create a resource model that would allow different roles to be distinguished within the configuration process. Which roles did they have in mind, exactly? Well, there are three primary roles defined in the Gateway API:

- **Infrastructure Provider** – responsible for Kubernetes cluster provisioning.
- **Cluster Operator** – responsible for managing the cluster once it has been provisioned (this includes managing policies, permissions, etc.)
- **Application Developer** – responsible for application configuration and Service composition.

When it comes to configuring the resources defined within the Gateway API their tasks would be as follows:

- The Infrastructure Provider creates GatewayClasses (they describe types of load-balancing/proxying infrastructure that can be provisioned in the cluster).
- The Cluster Operator creates Gateway instances derived from GatewayClasses (defined by the Infrastructure Provider).
- The Application Developer creates xRoutes (attached to existing Gateways) in order to define how traffic should be routed to applications.

Functionality

As mentioned, the Gateway API **covers a much wider set of functionalities** than Ingress. First, it provides support for various L4-L7 protocols, not just HTTP(S) as Ingress does. For each protocol a dedicated xRoute resource is defined (offering specific capabilities):

- HTTPRoute (for HTTP)
- GRPCRoute (for gRPC)
- TLSRoute (for TLS)
- TCPRoute (for TCP)
- UDPRoute (for UDP)

Second, compared to Ingress, the **set of HTTP(S) functionalities is also much richer**. In particular, the following features are covered:

- path-based matching
- header-based matching

- query parameters-based matching
- method-based matching
- request and response headers modification
- request mirroring
- request REDIRECT
- URL rewrite

Implementations

As is the case with Ingress, the Gateway API **does not provide any default controller** implementation built into Kubernetes. You are free to choose one of several available implementations. It is no big surprise that these are usually implementations offered by the same vendors that provide Ingress controllers or service mesh solutions. The list of existing implementations and their status can be found here. You should also know that individual implementations generally do not have to support all the features that are defined within the Gateway API. This is because so-called support levels have been introduced, assigned to individual features. Gateway API offers three support levels:

- **Core** – features assigned to this category will be portable and they are supposed to be supported by all implementations eventually.
- **Extended** – these features will also be portable but support is not mandatory.
- **Implementation-specific** – these features will not be portable and are vendor-specific.

Ingress vs. Gateway API – example use case

To understand what Gateway API can offer, let's consider an example use case. Imagine you are building a **system for observability**. It is a modular application in which individual components perform different tasks. You want to deploy this system in Kubernetes. **Two application modules are supposed to be accessible from the public network**. The first is a module for graphical data visualization, offering tools for statistical data analysis and data presentation (in the form of various types of charts and dashboards). The second module is the API to the data collector. This way, access to raw data is ensured as well.

A simple scenario

Here's how you can configure this using the well-known Ingress object (see Fig. 6).

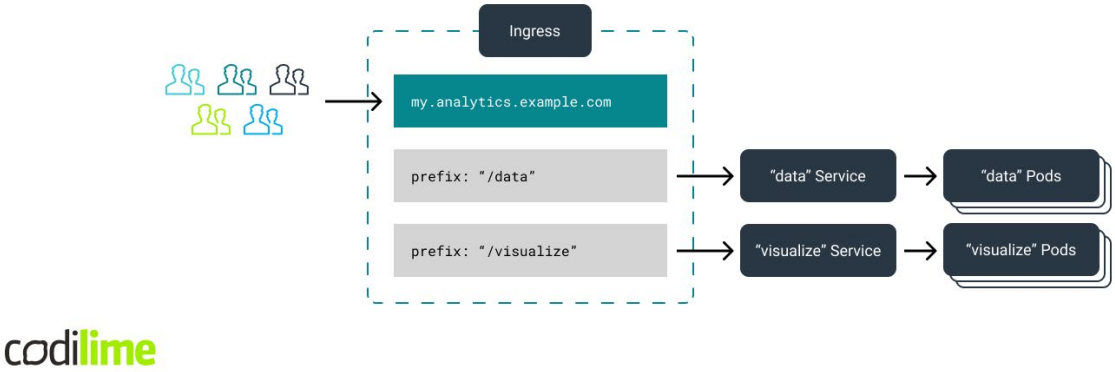


Fig. 6 Ingress-based configuration scheme

We assume that the routing to two application modules reachable from outside the cluster (e.g. from the Internet) takes place at the HTTP layer and is based on URL paths. For the "/data" prefix, HTTP requests to "my.analytics.example.com" will be forwarded to Pods representing the data collector's API module. For the "/visualize" prefix, the requests will go to Pods responsible for data presentation and graphical visualization. Of course, we assume that each Pod group has a dedicated Kubernetes Service instance created. The Ingress instance uses these Services (`.spec.rules[].http.paths[].backend.service`) to forward traffic to these Pod groups. The complete Ingress configuration is shown below.

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: ingress1
spec:
 rules:
 - host: my.analytics.example.com
   http:
     paths:
     - path: /data
       pathType: Prefix
       backend:
         service:
           name: data
           port:
             number: 8000
```

```
- path: /visualize
        pathType: Prefix
        backend:
          service:
            name: visualize
            port:
              number: 9000
```

ingress.yaml

Now let's use the Kubernetes Gateway API to reflect the above configuration (see Fig. 7).

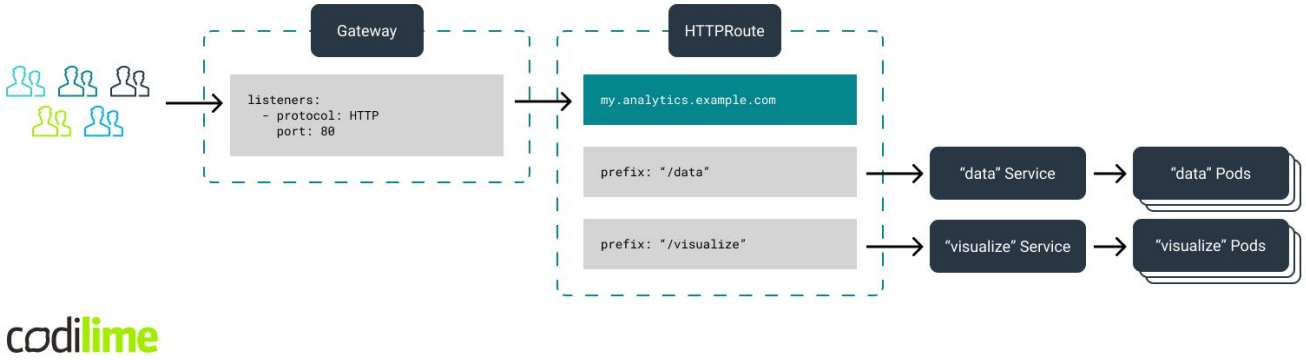


Fig. 7 Gateway API-based configuration scheme

The starting point is to create a GatewayClass to indicate which controller will be used underneath. Next, in the context of traffic processing, you need to use two objects defined within the Gateway API. The first is a Gateway instance listening for a certain type of traffic. Within its configuration, you define a single listener that listens to HTTP traffic on port 80. Next, you need to create an HTTPRoute instance. You specify a single entry "my.analytics.example.com" under the `.spec.hostnames[]` array and then you provide a set of rules (under `.spec.rules[]`) for HTTP requests matching such a host name. Similar to Ingress, you also configure path-based HTTP routing here. You can refer to individual groups of Pods (representing modules of your application) through the Service instances corresponding to them. You indicate these under `.spec.rules[].backendRefs[]`. Once you have this part of the configuration ready, all you have to do is to attach your HTTPRoute instance to one of the existing Gateways. You can do this in the `.spec.parentRefs[]` section where you point to the name of the Gateway you created. The entire configuration is presented below.

```yaml
---
kind: GatewayClass
apiVersion: gateway.networking.k8s.io/v1beta1
metadata:
 name: my-controller
spec:
 controllerName: my.company.example.io/my-gateway-controller

---
kind: Gateway
apiVersion: gateway.networking.k8s.io/v1beta1
metadata:
 name: web-gateway
spec:
 gatewayClassName: my-controller
 listeners:
   - name: http
     protocol: HTTP
     port: 80
     allowedRoutes:
       namespaces:
         from: All

---
kind: HTTPRoute
apiVersion: gateway.networking.k8s.io/v1beta1
metadata:
 name: httproute-basic
 labels:
   scenario: basic
spec:
 parentRefs:
 - group: gateway.networking.k8s.io
   kind: Gateway
   name: web-gateway
 hostnames:
 - “my.analytics.example.com”
 rules:
 - matches:
   - path:
       type: PathPrefix
       value: /data
```

```
    backendRefs:
    - name: data
      port: 8000
  - matches:
    - path:
        type: PathPrefix
        value: /visualize
    backendRefs:
    - name: visualize
      port: 9000
```

gatewayAPI-basic.yaml

A more demanding scenario

At this point, you might be thinking: it is nice that I was able to use Gateway API for my application, but why do I actually need it when I can accomplish exactly the same using Ingress? Well, you are basically right. However, you must remember that the scenario was relatively simple. All you needed was the efficient path-based HTTP routing that Ingress offers out of the box.

Now think about a slightly more complex scenario. Suppose you want to deploy a brand new version of the data collector API within your application. It is supposed to be an extended version of this module, which is divided into sub-modules, each being a separate API for metrics, traces and logs respectively. At the same time, you want to keep offering your users the old version of this module. But you want to limit its functionality a bit. First, you decide that they will only be able to access certain subsets of the metrics, not the complete data. Second, you don't want the API to allow users to perform CRUD operations, but instead offer read-only functionality. You have also decided that only a selected group of users will gain access to the new API. Such verification will be based on the value of cookies in the header of HTTP requests incoming to the old API. For a given value of cookies, a REDIRECT containing URL to the new API has to be made.

As for the module for data presentation and visualization, here you also plan some modifications, but not so extensive. You just want to add a few minor features to the current release. To test the stability of this latest version of the module, you want to use an approach based on the canary deployment strategy, where both versions of the module will run in parallel, but the latest one is allowed to handle only a small percentage (e.g. 5%) of all requests coming from users.

If you start analyzing this scenario from the technical point of view, you will find out that it requires the following mechanisms to be executed:

- **path-based HTTP matching** (as you still want to use URL prefixes to distinguish between traffic destined for the data collector API module and data visualization module).

- **header-based HTTP matching** (since you want to use cookies values for matching).
- **method-based HTTP matching** (since you plan to allow only the HTTP GET method to be executed on the old data collector API module).
- **request REDIRECT** (your scenario requires a REDIRECT to be executed).
- **weighted load balancing** (since you need non-symmetrical traffic split for data visualization module versions).

Now ask yourself: **which of these functions does Ingress cover?** The correct answer is: only the first from the above list. Therefore, Ingress will not be suitable for your complex scenario. Unless you apply some custom Ingress controller implementation that offers advanced functionalities. Remember, however, that then you will not really be using the native version of Ingress, but the CRDs and annotations instead, which are actually something extra.

Now it's time to ask the same question again, but now in the context of Gateway API: **which of the functions required in your scenario does Gateway API cover?** The correct answer is: all of them. Here is what it might look like (see Fig. 8).

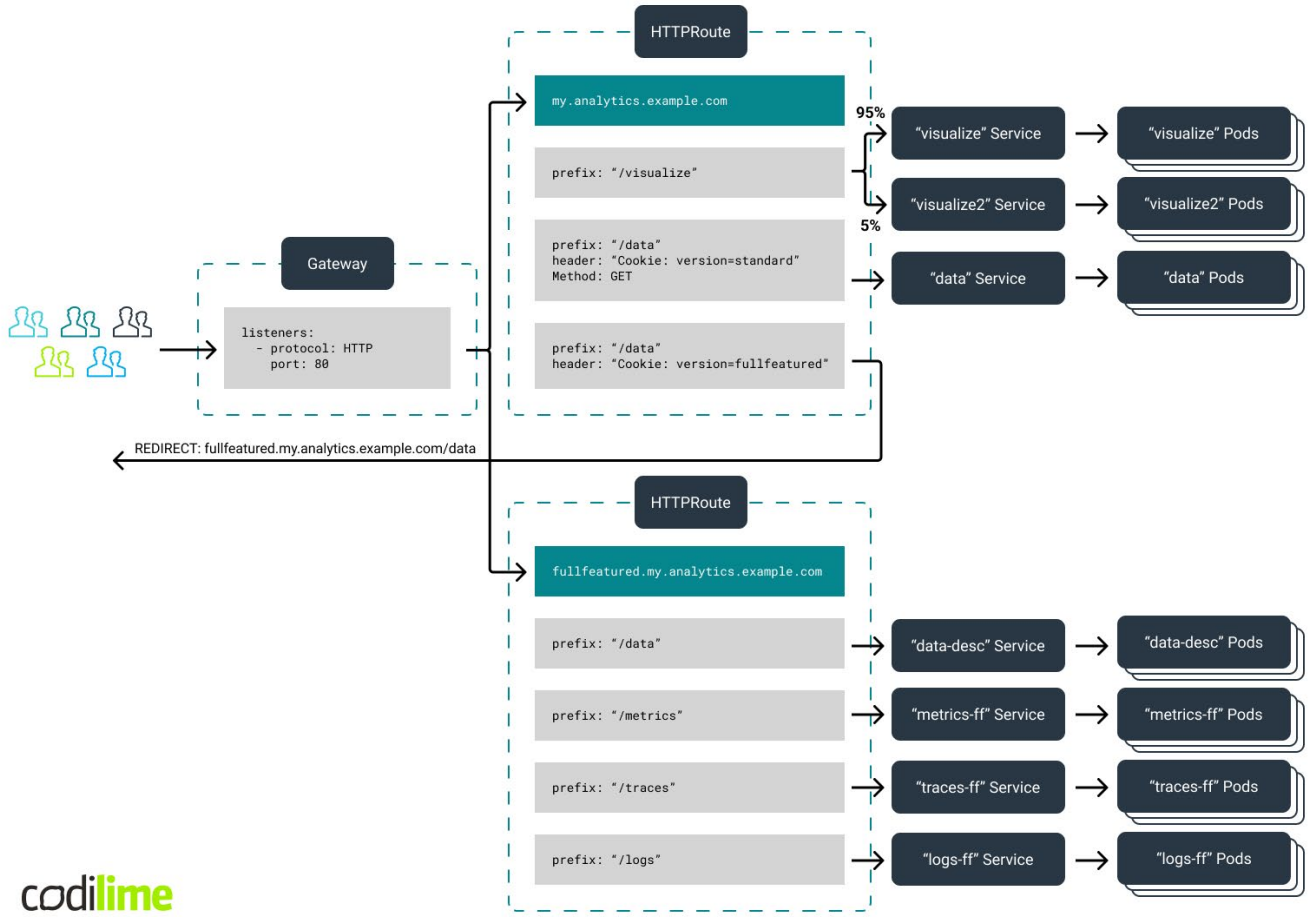


Fig. 8 Gateway API-based configuration scheme – advanced scenario

Here you can use exactly the same Gateway instance as for the basic scenario but now the previous HTTPRoute instance is replaced by two other routes attached to that Gateway. The first HTTPRoute instance will handle the requests to “my.analytics.example.com”. It has three rules configured:

- The first matches requests containing the “/data” prefix and the Cookie header value equal to “version=fullfeatured”. Such requests will not go to any backend. Instead, a REDIRECT will be executed to a new location equal to “fullfeatured.my.analytics.example.com/data”. Requests to the “fullfeatured.my.analytics.example.com” host will, in turn, be handled by a separate HTTPRoute instance. Routing to the backends corresponding to the individual sub-modules of the new data collector API is handled here based on the URL path prefix:
  - requests with “/data” prefix will be forwarded to the “pilot” web-page containing the URLs to the new APIs
  - requests with “/metrics” prefix will be forwarded to the metrics API
  - requests with “/traces” prefix will be forwarded to the traces API
  - requests with “/logs” prefix will be forwarded to the logs API
- The second rule matches only HTTP GET requests containing the “/data” prefix and the Cookie header value equal to “version=standard”. Such traffic will be forwarded to the simplified version of the old data collector API.
- The third rule matches requests containing the “/visualize” prefix. They will be forwarded both to the current version and the test (updated) version of the data visualization and presentation module. The traffic distribution will be based on the applied canary rollout strategy (95% to 5% respectively).

The entire configuration is presented below.

```
---
kind: GatewayClass
apiVersion: gateway.networking.k8s.io/v1beta1
metadata:
 name: my-controller
spec:
 controllerName: my.company.example.io/my-gateway-controller


---
kind: Gateway
apiVersion: gateway.networking.k8s.io/v1beta1
metadata:
 name: web-gateway
```

```
spec:
 gatewayClassName: my-controller
 listeners:
   - name: http
     protocol: HTTP
     port: 80
     allowedRoutes:
       namespaces:
         from: All


---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: HTTPRoute
metadata:
 name: httproute-advanced
 labels:
   route: advanced
spec:
 parentRefs:
 - group: gateway.networking.k8s.io
   kind: Gateway
   name: web-gateway
 hostnames:
 - my.analytics.example.com
 rules:
 - matches:
   - path:
       type: PathPrefix
       value: /data
     headers:
     - type: Exact
       name: Cookie
       value: "version=fullfeatured"
   filters:
   - type: RequestRedirect
     requestRedirect:
       hostname: fullfeatured.my.analytics.example.com
       statusCode: 302
```

```
  - matches:
    - path:
        type: PathPrefix
        value: /data

      headers:
      - type: Exact
        name: Cookie
        value: “version=standard”
      method: GET
    backendRefs:
    - name: data
      port: 8000
  - matches:
    - path:
        type: PathPrefix
        value: /visualize
    backendRefs:
    - name: visualize
      port: 9000
      weight: 95
    - name: visualize2
      port: 9000
      weight: 5


---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: HTTPRoute
metadata:
 name: httproute-advanced-data
 labels:
   route: advanced-data
spec:
 parentRefs:
 - group: gateway.networking.k8s.io
   kind: Gateway
   name: web-gateway
 hostnames:
 - fullfeatured.my.analytics.example.com
```

```
rules:
- matches:
  - path:
      type: PathPrefix
      value: /data
  backendRefs:
  - name: data-desc
    port: 80
- matches:
  - path:
      type: PathPrefix
      value: /metrics
  backendRefs:
  - name: metrics-ff
    port: 8001
- matches:
  - path:
      type: PathPrefix
      value: /traces
  backendRefs:
  - name: traces-ff
    port: 8002
- matches:
  - path:
      type: PathPrefix
      value: /logs
  backendRefs:
  - name: logs-ff
    port: 8003
```

gatewayAPI-advanced.yaml

Summary

This publication presents three solutions under the broad heading of application networking in Kubernetes. The first is **Ingress**, which performs HTTP(S) routing. Ingress allows you to handle north-south traffic, i.e. the traffic entering the cluster from outside which is then routed to various workloads deployed in the cluster. It is worth knowing that native Ingress is a relatively simple specification.

However, it has a worthy successor in the form of **Gateway API**, which allows you to express a wide set of advanced functionalities in a standard way. On the other hand, there might be scenarios where you need to control east-west traffic, i.e the internal traffic between workloads running in a cluster. This would be the case for a microservices-based application (deployed in the cluster) that contains many modules. Then, employing a **service mesh** solution to control such intra-application traffic in a sophisticated manner, and to gain deep insight into the communication between modules, will be a reasonable approach.

Application networking solutions in Kubernetes are evolving. This applies not only to the emergence of Gateway API. The solutions from the service mesh family are also being developed with both new functionalities on board and better performance. This is understandable because Kubernetes as a platform is becoming more and more ubiquitous and it needs better and better solutions in many areas, including the ones described in this publication. Solutions architects, application developers and product owners should definitely be up to date with the modern solutions for application networking.



About the author

Paweł Parol – Solutions Architect

Paweł is a solutions architect with many years of experience in the telco/IT industry. He is an experienced engineer who has worked in both operational and research projects throughout his career. Currently, he is a tech lead in CodiLime's R&D department where he creates solution architectures for PoC projects that integrate many emerging technologies into larger solutions. His main areas of interest include cloud architecture, Kubernetes, and cloud-native solutions, as well as virtual networks.

About CodiLime

Since 2011, CodiLime has been the engineering partner of choice for semiconductor companies, networking vendors, telecom services, and software solution providers. Our experts help explore their data assets better by creating scalable solutions that make storing, processing, and analyzing data a breeze.

We appreciate long-term collaborations above all, as well as our partners. Below are some of the names that have already trusted us:

CodiLime aims to **link network engineering talent with business domain expertise** – we focus on five N.E.E.D.S. – Networks, Equipment, Environment, Data and Security.

Do you need any further information regarding our services? Don't hesitate to contact us!

contact@codilime.com