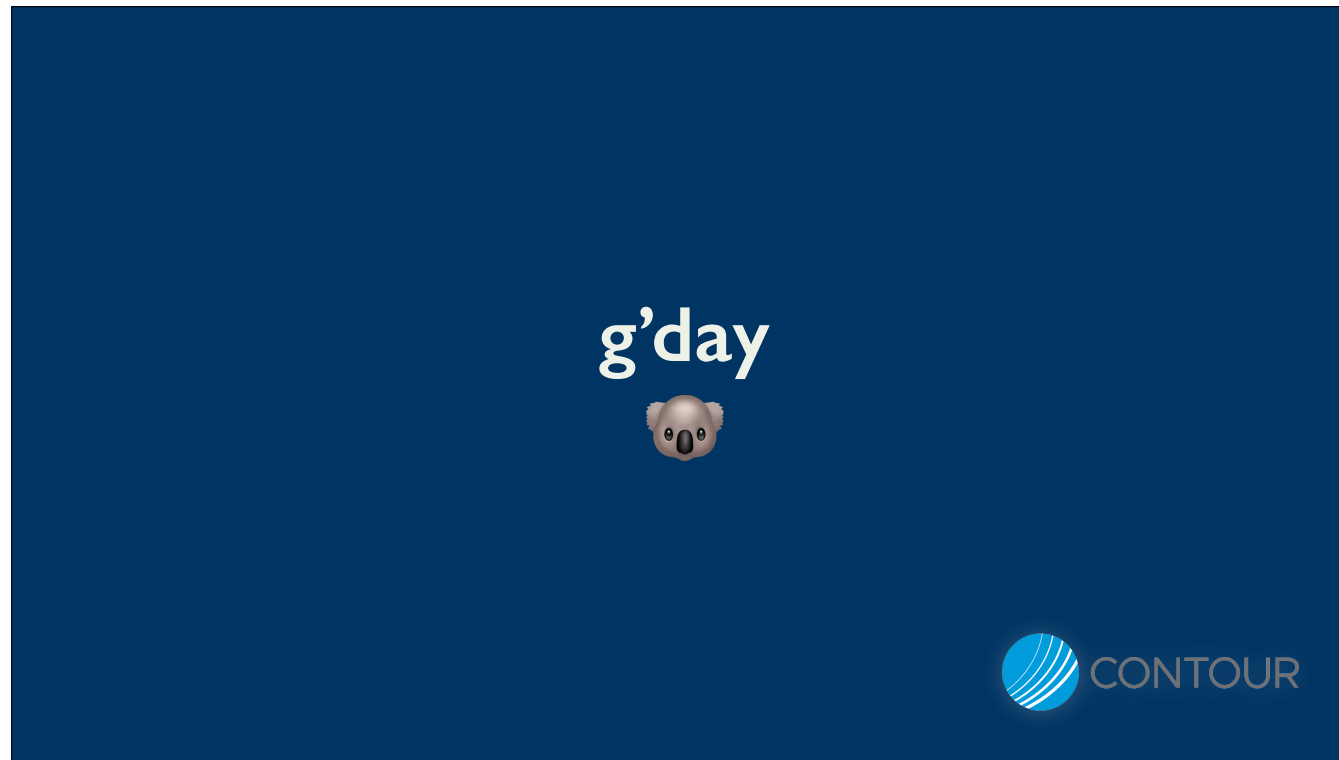


# Addressing the limitations of Kubernetes' Ingress object

David Cheney  
Staff Engineer, VMware





Good morning

My name is David, I'm a software engineer visiting Shanghai from Sydney, where I work for VMware.

I'm the tech lead on a product called Contour, an kubernetes ingress controller using Lyft's Envoy as our data plane.

# Ingress-what controller?



The part of kubernetes that I spend my time in is something called an ingress controller.

Ingress controllers are responsible for getting traffic from the outside world down to your pods.

In practical terms this means; HTTP, TLS, load balancers, reverse proxies all at Layer 7.

To be clear, although we use Envoy for our data plane, Contour is not an service mesh. This is a common misunderstanding

We're focused exclusively on the problems of bringing remote traffic into your cluster.

**A good ingress controller should  
take care of the 90% use case for  
deploying HTTP middleware**



That's quite a broad remit so the way I approach the design space is I think that a good ingress controller should take care of 90% of the cases that traditionally you would have used an apache, or nginx, or squid, sidecar container or middleware, something in the request flow to augment your app.

# Getting to the 90% case

- Traffic consolidation
- TLS management
- Abstract configuration
- Path based routing



Here are some things that I think contribute to an ingress controller getting to that 90% level of functionality.

The first one is consolidation. If you use a service load balancer then every service you deploy has an ELB in front of it, that's a cost, and also a maintenance issue. They consume a public IP which are a scarce resource, and depending on your company your security team may not be cool with hundreds of public IPs funnelling traffic into their kubernetes cluster.

The second is TLS management. It's 2019, you need to be talking TLS. There is a move by Chrome and other browsers to mark non https sites "insecure". On the k8s side we have projects like cert manager and lets encrypt that take care of obtaining-and maintaining-a certificate, and an ingress controller covers presenting that certificate on port 443, so there shouldn't be any reason to not be secure in 2019. We have the technology

The third is a notion of being able to describe the properties of your web application in an abstract manner, or at least to have some portability between different ingress controllers (and clouds). You should be able to talk about the host name, tls configuration, route names and backends for those routes without having to write an apache configuration, or an nginx configuration.

Path based routing; with a service, all the traffic goes straight to the cluster IP, if you wanted to serve your static images from one service, and your dynamic data from another, you can't do that with a service ip, you would have to stand up some kind of reverse proxy to handle the traffic.

# What is Contour?



So what is contour?

Contour is an ingress controller that I've built at Heptio, now VMware.

Contour exists to fulfil the requirements I just described.

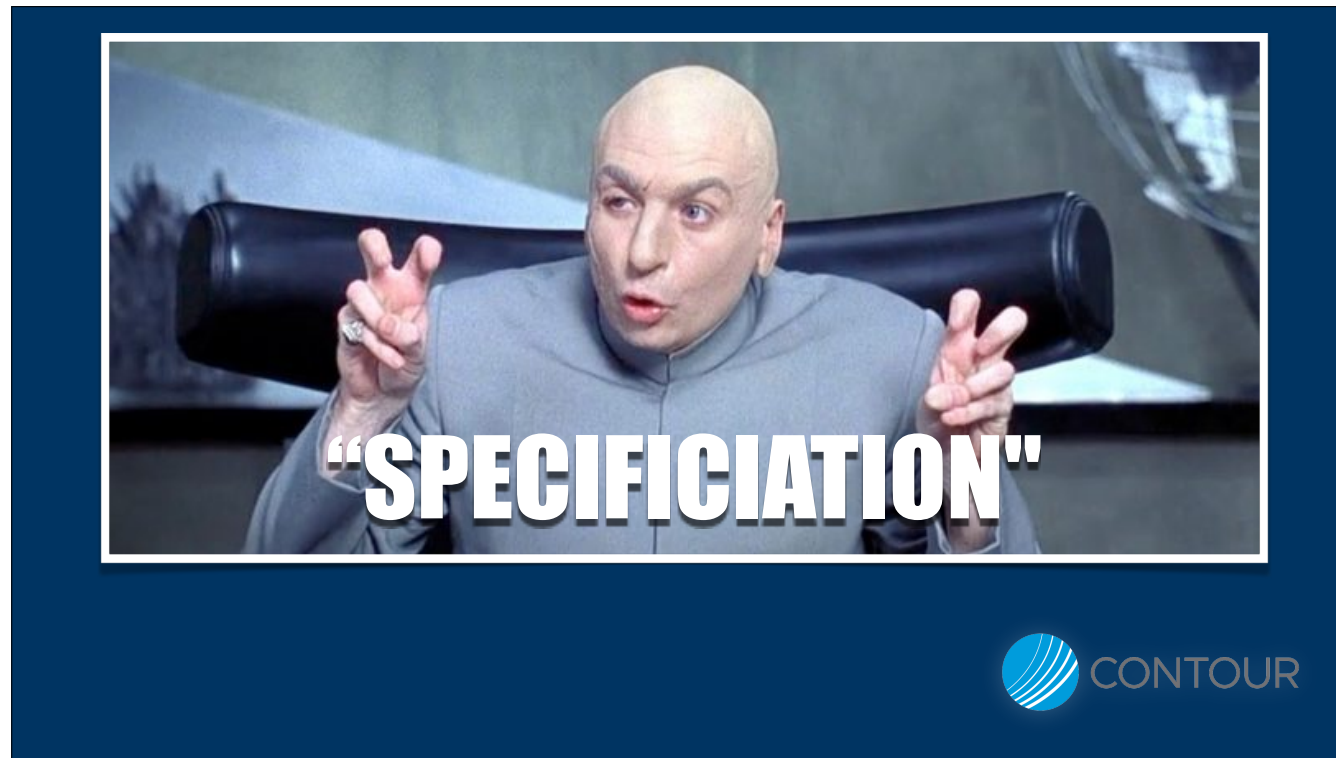
I want to be clear that this talk isn't a product pitch — well it is — but it's not for contour.  
The fact that the stuff I'm going to talk to you about is implemented in contour is incidental

# What are the problems with Ingress?



Instead what I want to talk about is something that is larger than any of the ingress controllers out there in the market. And that is the ingress object.

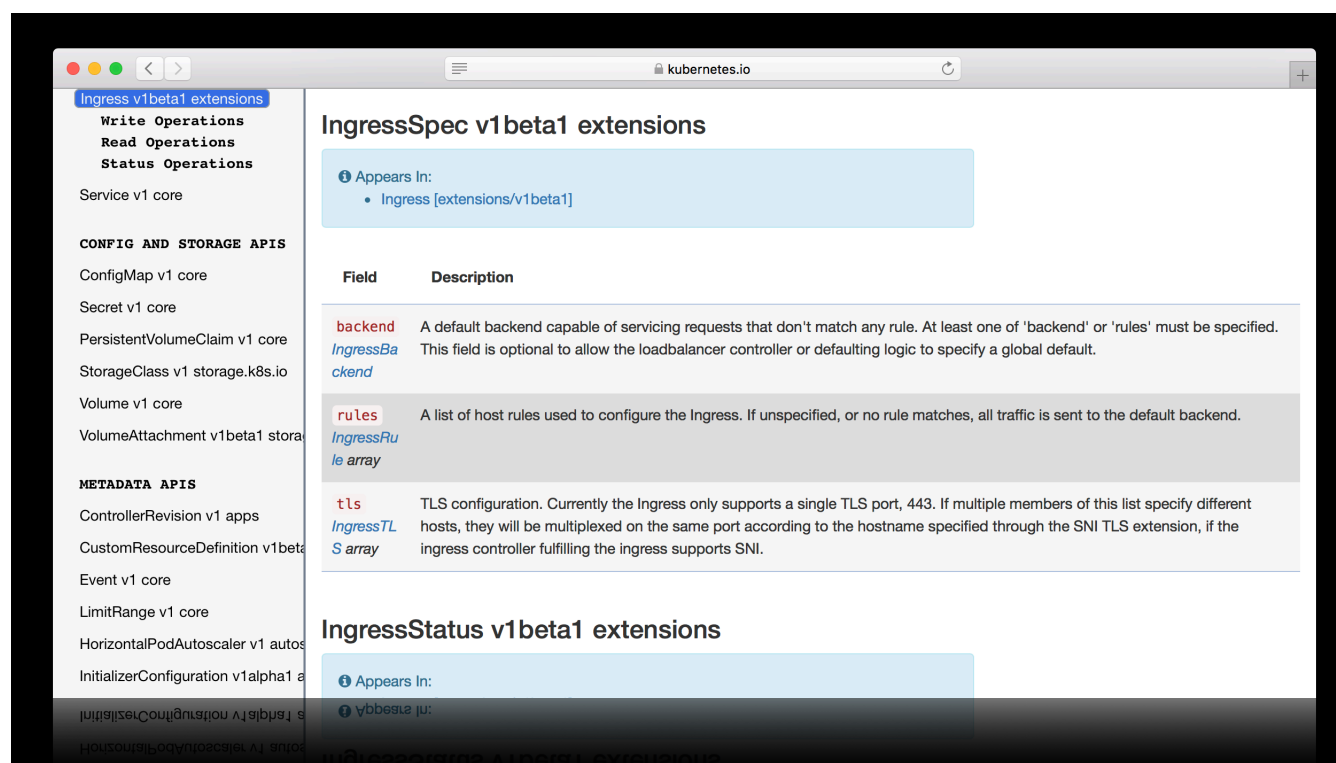
With that background, let's talk about the problems with the current Ingress object.



I think my biggest complaint with ingress is the “specification” is a bunch of text in comments on the api data structure.

JSON is not a specification; it’s barely a schema.





This is literally the specification implementors have to work from.

This is perfectly fine for someone who needs a schema to send a message to the k8s API server, if you're hacking around with curl, or something like that.

But as an implementor, or someone building tools on top of this API, we need a level of formality in this specification simply is not present.

What I would like to see is something akin to the level of detail of an internet RFC, because, as we'll see, the ingress 'specification' is rife with ambiguity,

# Gosh darned default backend



Take for example the default backend.

“A default backend capable of servicing requests that don't match any rule.”



This is what the “spec” says about the default backend. This is `_all_` it says.

What we have is a situation where each ingress document has a notion of default backend. Requests that don't match any rule are routed to the default backend

This would make sense if there is only ever one single ingress document, but that is almost never the case, also as we'll see routes can fail to match for reasons other than the path.

So this idea of a default backend is not a catch all route on your vhost, its something else. But that something is left to the interpretation of the implementor, to the detriment of ingress users expecting portability.

# Default backend ambiguity

- Default backend conflates the notion of a vhost, the Host: header traffic arrives on, from the backend to serve it
- The host key in spec.rules is *optional* — does this mean the rule matches *any* host?  $\neg\_(\emptyset\_ \emptyset)\_/\_$
- Default backend can be present in multiple Ingress objects — which takes precedence?



Default backend conflates ...

A default *vhost* is a notion of a http handler for traffic that fails all other routing rules — that needn't be a backend, it could be a simple 302 redirect

Host key is optional ...

Default backend present in multiple—all?—ingress objects, which takes precedence? Should they be merged together? That's a bit tricky because the default backend is a service, not something you can do a route match on.

And this ambiguity suggests that a default backend can be namespaced because ingress objects can be namespaces, but of course that doesn't work.

# Ingress objects can span namespaces



Speaking of namespaces, the ingress spec permits the definition of a virtual host to span more than one ingress object. This means that the definition of a virtual host can span namespaces.

I can see the argument for this; ingress objects can only refer to services in their namespace, but what if you had gone down a per team namespace model? You may want to have /finance managed in the finance namespace and /ads in the ads namespace? You do this by putting part of the vhost definition in the finance namespace and part of the vhost definition in the ads namespace. They both refer to the same virtual host, so the ingress controller stitches them all together for you.

However, this means that if someone has RBAC permission to add an ingress object in their namespace, they can inject a route onto the ingress you defined in your namespace even if they never had permission to edit your ingress route!

**Cert-manager *relies* on this  
feature to support Let's  
Encrypt's HTTP-01  
challenge!**



And just in case you were thinking — hmm this sounds like a massive security hole, I'd like to disable this please — you cannot because in a perfect example of Hyrum's Law, projects like cert-manager rely on this ability to inject a route onto your vhost from the cert-manager namespace so they can route the HTTP-01 challenge to a service running in their namespace

**Ingress makes shared tenancy  
difficult if your tenants aren't  
incentivised to play nicely with  
each other**



What this boils down too is Ingress is very difficult to use in a shared kubernetes cluster.

There are no safeguards to prevent anyone with RBAC permission to create or edit ingress objects from accidentally, or maliciously, injecting conflicting or invalid configuration onto the vhost for another tenant.

# One route. One Service.



Lets talk about some other problems with the ingress spec that affect people trying to use the modern web application patterns.

Kubernetes services are mapped onto http routes via an ingress document, however the ingress spec only permits `_one_` service per route.



IngressBackend describes all endpoints for a given service and port.



Appears In:

- [HTTPIngressPath extensions/v1beta1](#)
- [IngressSpec extensions/v1beta1](#)

Field	Description
<code>serviceName</code> <i>string</i>	Specifies the name of the referenced service.
<code>servicePort</code>	Specifies the port of the referenced service.

Now a kubernetes service can match multiple pods if they share the same label, they'll all get mixed into the same endpoint document, but at best you're going to get a weighted distribution across the deployments that make up the service.

If you want to do a canary deployment and send 1% of your traffic to the new version of your application, you'll need to have 99 pods running the old version of your app to make the ratios work out.

# Annotation potpourri



Another big problem with the ingress spec is the schema is so limited the only place you can stuff extra parameters or attributes about your web application is in annotations

## Cambrian explosion of Ingress annotations

- Allow port 80 and/or 301 upgrade to HTTPS
- Request timeout (applies to all the entries in the Ingress document)
- Retry parameters (also applies to all entries in the Ingress document)
- TLS minimum protocol version
- Websocket enabled routes



This has led to a Cambrian explosion of ingress document annotations.

here are just some that contour implements, this is barely scratching the surface of what has been shoehorned into an untyped map of annotations by various ingress controllers.

If this Lasso fair approach wasn't bad enough, the configuration of a vhost may be spread across several ingress documents, so how these annotations are applied is not obvious and often confusing.

[ 3 click ]

eg. 301 upgrade settings, request timeouts and retry parameters likely apply per ingress document not per vhost, therefore if you want those settings to apply to some routes and not to others for a vhost, you have to split them across two ingress documents

TLS minimum protocol version has to be specified in an annotation because the TLS stanza of the ingress document has no place for it; same with cipher specs.

if you split a host across several ingress documents, do things like TLS min protocol apply to all the ingresses that match that host, or just the one where the annotation is present? There is no right answer here. If you say TLS min proto applies only to the ingress spec in a single document, you're committing to altering the TLS handshaking operation based on the request line of the HTTP request which you don't have at that point.

If you say that annotations like TLS min proto apply across any ingress with that host, because ingresses can span namespaces, someone in another namespace can alter the TLS parameters for your virtual host even though they don't have permission to write into your namespace.

The problem isn't hard; it has no solutions.

# Ingress isn't broken, it's just limited



This is a good time for me to pause to say that while I personally have a bunch of gripes with Ingress coming from my position as an implementor, ingress isn't broken.

I don't want you to come away from this talk thinking "welp, dave says I can't use that at all"

Ingress isn't broken, it's just limited, and if you're not hitting those limits then far be it from me to tell you need to change what you're doing.

# What is Contour going to do about these problems?



However if you have experienced some of these problems, then let me tell you about what we're doing in contour to try to improve the situation.

At the start of the 2018, when we were still called Heptio, we signed a joint development deal with Yahoo Japan to build them a large load balancing solution for them using a kubernetes cluster almost like an appliance.

# Ingress IngressRoute



Realising that yahoo Japan were encountering many of the issues with multi tenancy that I described above we set out to define a new kind of ingress document, which we call ingressroute

**Every IngressRoute document  
has one hostname**



The first thing that we changed is each ingress route document refers to one hostname and one hostname only.



```
apiVersion: contour.heptio.com/v1beta1
kind: IngressRoute
metadata:
  name: blog
  namespace: marketing
spec:
  virtualhost:
    fqdn: blog.heptio.com
    tls:
      secretName: blog-secret
  routes:
  - match: /
    services:
    - name: blog-svc
      port: 80
```

**The virtualhost key indicates  
this is a root ingressroute**



This means all the properties of a virtual host, its name, its tls parameters, the secret that holds the tls certificate are in one namespace alone.

We call this the ingress route document a root, for reasons I'll explain in a little bit

## Load balancing strategies can be specified per backend service



Because we're no longer limited to the schema of the kubernetes ingress object we now have a place to hang configuration attributes that used to be smuggled into annotations.

```
apiVersion: contour.heptio.com/v1beta1
kind: IngressRoute
metadata:
  name: blog
  namespace: marketing
spec:
  virtualhost:
    fqdn: blog.heptio.com
    tls:
      secretName: blog-secret
  routes:
  - match: /blog
    services:
    - name: blog-svc
      port: 80
      strategy: WeightedLeastRequest
```

For this route, use  
**WeightedLeastRequest**  
across the endpoints  
matching blog-svc



For example, per route, per service, you can control the load balancing strategy that will be used across the endpoints that make up this service.

# Websocket support



A key reasons for choosing Envoy as our data plane was Envoy's support of long running websocket sessions across configuration changes.

```
apiVersion: contour.heptio.com/v1beta1
kind: IngressRoute
metadata:
  name: chat
  namespace: default
spec:
  virtualhost:
    fqdn: chat.example.com
  routes:
    - match: /
      services:
        - name: chat-app
          port: 80
    - match: /websocket
      enableWebsockets: true
      services:
        - name: chat-app
          port: 80
```

**Only permit  
Upgrade: websocket  
on /websocket**



Enabling websocket support per route is as simple as adding the `enableWebsockets: true` key to your route.

In general where something can be enabled for all use cases rather than having a parameter or flag that people have to know to turn on, my policy is to turn it on across the board; http compression is a good example of this.

However I wasn't comfortable permitting `Upgrade: websocket` by default for all routes, so we made a deliberate decision to make it opt in only.

# Per route Timeout and Retries



Another common feature which previously was smuggled into an annotation are timeouts and retries

```
apiVersion: contour.heptio.com/v1beta1
kind: IngressRoute
metadata:
  name: request-timeout
  namespace: default
spec:
  virtualhost:
    fqdn: timeout.bar.com
  routes:
  - match: /
    timeoutPolicy:
      request: 1s
    retryPolicy:
      count: 3
      perTryTimeout: 150ms
    services:
    - name: s1
      port: 80
```

**Total time to service request  
one second**

**Try three times, waiting  
150ms each time**



# Multiple service backends



The kubernetes ingress document limits routes to a single backend service. Using ingressroute we have the ability to say instead of a single service, allow a list of services.



```
apiVersion: contour.heptio.com/v1beta1
kind: IngressRoute
metadata:
  name: blog
  namespace: marketing
spec:
  virtualhost:
    fqdn: blog.heptio.com
    tls:
      secretName: blog-secret
  routes:
  - match: /
    services:
    - name: service1
      port: 8080
    - name: service2
      port: 8080
```

**Traffic will be load balanced  
across service1 and service2**



# Weighted services



The main reason you'd want to have more than one backend service per route is to enable patterns like canary deploys or blue/green deployments

```
apiVersion: contour.heptio.com/v1beta1
kind: IngressRoute
metadata:
  name: gmail
  namespace: google
spec:
  virtualhost:
    fqdn: gmail.google.com
  routes:
  - match: /
    services:
    - name: gmail-v1.3.1
      port: 80
      weight: 90
    - name: gmail-v2.0.0
      port: 80
      weight: 10
```

Shift traffic from  
v1.3.1 to v2.0.0 by altering  
the service weights



In this example, 90% of the requests to gmail.google.com are routed to the version 1.3.1 and 10% are routed to version 2.0.0. As you gain confidence in the the deployment you can edit the document to increase the weighting towards version 2.

And of course, weighting, load balancing strategy, websockets, etc can be combined per service, per route, depending on your applications needs.

It's important to note that modifying any of these properties is immediately transmitted to envoy; there's no delay, no process reloading, no SIGHUP

# Delegation



Delegation is our answer to helping multi tenant clusters stay manageable.

All the ingressroute documents we've seen so far are what we call "root documents", because they are at the root of a delegation tree.

To explain why we think delegation is powerful let me lay out a scenario for you.

# https://google.com/finance

- You want to delegate control of https://google.com/finance to the Google Finance developers working the google-finance namespace.
- The Google Finance team should not be able to alter the configuration for the rest of https://google.com/
- None of the teams working on https://google.com/ should have access to the TLS secret for https://google.com/



[ read ]

Just think; how would you do this securely in a k8s cluster today?

You can certainly split the routes across various namespaces, but as I've laid out, the rest of the ingress spec severely undercuts the security this deployment

```
apiVersion: contour.heptio.com/v1beta1
kind: IngressRoute
metadata:
  name: google-com
  namespace: google
spec:
  virtualhost:
    fqdn: google.com
    tls:
      secret: google-com-secret
  routes:
    - match: /
      delegate:
        name: search
        namespace: google-search
    - match: /finance
      delegate:
        name: finance
        namespace: google-finance
```

The configuration for /finance  
is found in the  
google-finance/finance  
ingressroute




In this example we have a standard ingressroute root; its for google.com, and references google-com-secret in the google namespace.

However all the routes, / and /finance refer to ingressroute documents in other namespaces. You can think of this as an “include” macro, contour will find the configuration fragment for the search service in the search namespace, and the finance service in the finance namespace.

Let's have a look at the finance document in the google-finance namespace

```
apiVersion: contour.heptio.com/v1beta1
kind: IngressRoute
metadata:
  name: finance
  namespace: google-finance
spec:
  routes:
  - match: /finance
    services:
      name: finance-v1.0.1
      port: 8080
```

Only routes that are a sub match of /finance



Here is the finance ingressroute document.

It does not have a virtual host stanza, which means it is not a root. It is a delegate and can only be referenced by other ingressroute documents that delegate to it explicitly. And contour is only going to reference routes that start with the prefix that was delegated too

DNS example

# Restricted root namespaces



The final piece of the multi tenant puzzle is the ability to restrict the namespaces that contour will look for ingressroute roots.

This is an opt in feature, by default anyone with RBAC permission to create an ingressroute can do so, but if you want to make creating a new root a administrative event, you can configure contour to only look for roots in a set of namespaces which you have arranged that only administrators can write too.



Thank you!

👉 [github.com/heptio/contour](https://github.com/heptio/contour)

👉 [#contour](#) on the k8s slack

👉 [cheneyd@vmware.com](mailto:cheneyd@vmware.com)



*Image: Egon Elbre*

Thank you for your time.

If you want to talk to me about anything I've said here, ingress, kubernetes, or go, come find me, I've got stickers for a company which won't exist in a week, so they might be worth something as antiques.