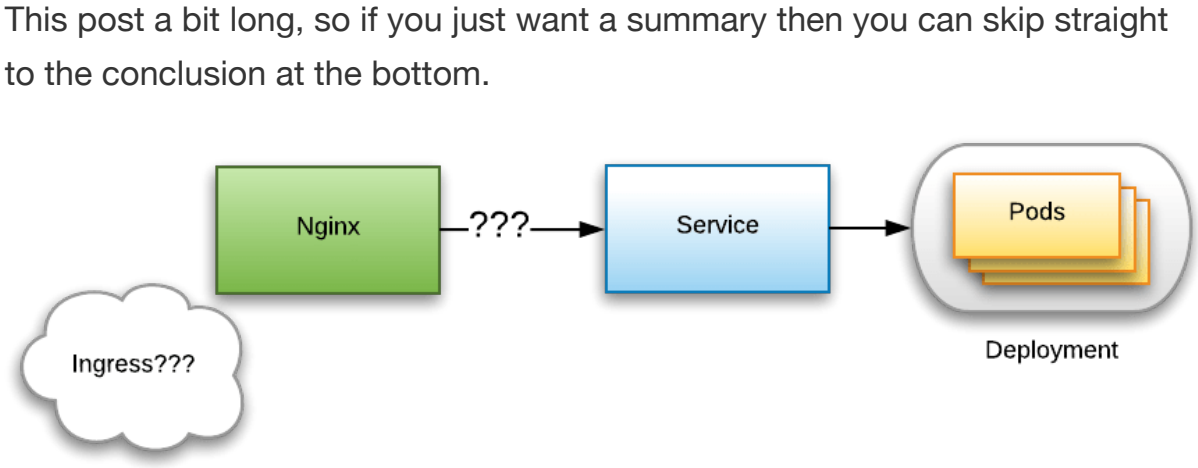
Studying the Kubernetes Ingress system

By Hongli Lai follow on twitter

Mar 26, 2018 · DevOps, Kubernetes, Research, Featured posts

I have been researching how the Kubernetes Ingress system works. My use case is to setup an autoscaled Nginx cluster that reverse proxies to Pods in multiple Deployments. It wasn't immediately obvious how to do this. By default, Pods in Kubernetes are not supposed to be reachable from outside the cluster. One makes them reachable either by associating those pods with a Service of the right type (i.e. either NodePort or LoadBalancer), or by defining an Ingress. But what *is* an Ingress? How do I put Nginx in between an Ingress and a set of Pods? This post describes my journey through the jargon-loaded Kubernetes documentation which does not hold any hands, as well as my journey through the Kubernetes source code, all in a quest to find answers.

This post a bit long, so if you just want a summary then you can skip straight to the conclusion at the bottom.



Defining a Service is straightforward enough... but how do you point Nginx to that? And how does Nginx relate to this Ingress thing?

How my journey began

Suppose I create a deployment for the 'echoserver' container on port 8080:

```
kubectl run hello-minikube --image=k8s.gcr.io/echoserver:1.4 --
port=8080
```

I'm not worried about autoscaling just yet. How do I reverse proxy Nginx to this set of pods? My first intuition was to expose this deployment with a service, after which I set up Nginx to reverse proxy to this deployment. But how do I find out the IP address of the deployment, how do I tell Nginx about it, and how do I reconfigure Nginx when the IP address changes?

I hypothesized that Kubernetes has some sort of API that I can query. After Googling for "Kubernetes Nginx" I found the folowing interesting resources:

- NGINX and NGINX Plus Ingress Controllers for Kubernetes Load Balancing by Nginx Inc.
- Setting up Nginx Ingress on Kubernetes – Hacker Nooon
- Source code of the community Kubernetes Ingress controller (kubernetes/ingress-nginx)
- Source code of the Nginx Inc. Kubernetes Ingress controller (nginxinc/kubernetes-ingress)

I had already heard about this Ingress thing, and I believed that an Ingress deploys a load balancer from the cloud provider on which Kubernetes is deployed. These search results suggest that there are multiple implementations of Ingress (which are categorized as *ingress controllers*), some of which apparently use Nginx. But how does that work? How does one activate a specific implementation? In any case, I believed that the key to finding out how to tell Nginx about my services'/pods' IP addresses (and thus how to query the Kubernetes API) lies in researching how the Nginx ingress controller works.

Installing the Nginx ingress controller

Before figuring out how the Nginx ingress controller works, I should first install it and see it action. According to the Ingress documentation and [the kubernetes/ingress-nginx documentation], I need to apply the following config:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: nginx-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    # Avoid the Google Compute Engine controller from processing
this Ingress.
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: hello-minikube
          servicePort: 80
```

I did this on my Minikube installation and then… nothing happened.

After some Googling I found out that Kubernetes does not do anything by default to process Ingress resources. One needs to manually deploy an ingress controller – this is the "picking an implementation" part. Apparently Google Cloud's Kubernetes Engine deploys a "Google Cloud controller" by default

which responds to Ingress resources and provisions a Google Cloud load balancers. The `kubernetes.io/ingress.class` annotation tells the Google Cloud controller to ignore that Ingress resource.

In addition, one needs to install an Nginx ingress controller – Minikube does not supply it by default. The kubernetes/ingress-nginx documentation describes deployment instructions for various infrastructures, and apparently on Minikube one is supposed to run this:

```
minikube addons enable ingress
```

Success! This resulted in a port being opened which I can access from my laptop. The usable URL could be fetched with:

```
minikube service nginx-ingress --url
```

The anatomy of an Nginx ingress controller installation

Now that I managed to install the Nginx ingress controller, it was time to find out how it looked like. What exactly did `minikube addons enable ingress` do?

In the Kubernetes dashboard I saw that I now have an `nginx-ingress-controller` replication controller. Further inspection and research revealed that Minikube applied the config files in its deploy/addons/ingress directory, with the most interesting one being `ingress-rc.yml`. That file deploys two replication controllers:

- The aforementioned "nginx-ingress-controller". This uses the image `quay.io/kubernetes-ingress-controller/nginx-ingress-controller:0.9.0`, which corresponds to the kubernetes/ingress-nginx Github repository, tag nginx-0.9.0.
- A "default-http-backend". This uses the image `k8s.gcr.io/defaultbackend:1.4`, which does nothing besides responding to `/healthz` (for implementing liveness/readiness probes) and returning 404 Not Found for all other URLs. Nginx-ingress-controller routes to this default-http-backend if a request does not match any of the known routing rules.

I `kubectl exec`'ed into the nginx-ingress-controller pod and found that it ran an Nginx installation. Looking into /etc/nginx, it appeared that the config file was somehow autogenerated. So an Nginx ingress controller *is* an Nginx instance at the same time, and the controller configures the internal Nginx installation to do the right thing.

Finding the container's entrypoint

The nginx-ingress-controller container runs the following command:

```
args:
  - /nginx-ingress-controller
  - --default-backend-service=$(POD_NAMESPACE)/default-http-backend
  - --configmap=$(POD_NAMESPACE)/nginx-load-balancer-conf
  - --tcp-services-configmap=$(POD_NAMESPACE)/tcp-services
  - --udp-services-configmap=$(POD_NAMESPACE)/udp-services
  # use minikube IP address in ingress status field
  - --report-node-internal-ip-address
```

What is this nginx-ingress-controller thing and where can I find its source code? I knew that I had to look in the kubernetes/ingress-nginx Github repository, tag nginx-0.9.0, but it wasn't immediately obvious from analyzing its Makefiles where the program's entrypoint is. So I grepped for `package main` – a directive in Go that signifies that this is a program's entrypoint – and found cmd/nginx/main.go.

From there, I followed the flow of the code in search for interesting things. The main function consists of fairly straightforward boilerplate things:

- CLI flags parsing: `showVersion, conf, err := parseFlags()`
- Initializing a client object for talking to the Kubernetes API: `kubeClient, err := createApiserverClient(conf.APIServerHost, conf.KubeConfigFile)`
- Starting a builtin HTTP server for querying the controller's status and debugging info: `go registerHandlers(conf.EnableProfiling, conf.ListenPorts.Health, ngx, mux)`

But it seems the bulk of the interesting stuff happens inside the "NGINXController" class:

```
ngx := controller.NewNGINXController(conf, fs)
...
ngx.Start()
```

The daunting NGINXController class

The NGINXController class is implemented in internal/ingress/controller/. The `nginx.go` file implements the `NewNGINXController()` and `Start()` methods.

The sources however quickly became daunting, and I found myself not understanding what's going on. This forced me to go on a sidequest. Fate led me to the Kubernetes sample controller, which demonstrates how a simple Kubernetes controller is written. I proceeded to study that.

The Kubernetes sample controller

The sample controller is based on the idea that its purpose is to process certain resources in the Kubernetes database. The sample demonstrates basic operations such as:

- How to register a new custom resource type.
- How to create, get and list resources of that type.
- How to respond to create/update/delete events.

Processing happens during controller startup, but also in response to create/update/delete events.

What does "processing" mean? According to the sample, its purpose is to compare the actual state of the resources with the desired state, and attempts to converge the two. It should also update the `Status` block of affected resources, per the Kubernetes API conventions.

In case of this sample controller, its goal is to ensure that for each `Foo` resource there is a corresponding `Deployment` of the `nginx` image. It also ensures that the replica count of that Deployment matches the replica count specified inside the Foo resource.

The entrypoint of the sample controller is `main.go`. It is very similar to `kubernetes/ingress-nginx`'s main.go: it sets up a bunch of stuff and calls `controller.Run()`. Most of the interesting stuff is in controller.go.

The Controller class's constructor sets up a few Informer objects, which are used to watch for create/update/delete events on resources of types `Foo` and `Deployment`.

When a `Foo` create/update/delete event is detected, it eventually calls `controller.syncHandler()`. This does not happen immediately or directly: such events are put into a rate limiting workqueue for two reasons:

1. To prevent the controller from being overloaded by too many events.
2. To serialize the processing of events, because Informers run in multiple background goroutines.

The code snippets below show how the Controller constructor sets up a workqueue and watches for Foo resource events. Whenever an event is detected, it calls `controller.enqueueFoo()`, which merely puts something in the workqueue.

```
/////// In NewController():
controller := &Controller{
    ...
    workqueue:
workqueue.NewNamedRateLimitingQueue(workqueue.DefaultControllerRateLim
 "Foos"),
    ...
}
...
fooInformer :=
sampleInformerFactory.Samplecontroller().V1alpha1().Foos()
// Set up an event handler for when Foo resources change
fooInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs

    AddFunc: controller.enqueueFoo,
    UpdateFunc: func(old, new interface{}) {
      controller.enqueueFoo(new)
    },
})

/////// In enqueueFoo():
c.workqueue.AddRateLimited(key)
```

The workqueue is processed by a background goroutine which, in an infinite loop, consumes the workqueue and calls `controller.syncHandler()` on the consumed item:

```
// In runWorker():
for c.processNextWorkItem() { }

// In processNextWorkItem():
obj, shutdown := c.workqueue.Get()
...
key, ok = obj.(string)
...
err := c.syncHandler(key)
```

`controller.syncHandler()` is where processing happens. The work it performs is as described at the beginning of this subsection.

Analyzing the NGINXController class, take 2

Armed with the newly gained knowledge on how controllers work, I went back to my main quest of figuring out how the Nginx ingress controller works. It turned out that many patterns from the sample controller occur also in the Nginx ingress controller. Fundamentally, the Nginx ingress controller listens for create/update/delete events on resources it is interested in, enqueues work into a rate limiting queue, and then a handler function fetches work from the queue in an infinite loop and processes each item.

The code that sets up the event listeners turned out to be in internal/ingress/controller/listers.go, function createListers(). It listens for events on the following resource types:

- Ingresses
- Endpoints
- Secrets
- ConfigMaps

The queue is represented by `controller.syncQueue` and the queue handler function is in internal/ingress/controller/controller.go, function syncIngress(). This function collects all necessary information to regenerate the Nginx config file: it fetches all relevant Ingress objects and looks up associated Pods' IP addresses that the Ingresses should route to.

`syncIngress()` then calls internal/ingress/controller/nginx.go, function OnUpdate() to actually write out the new Nginx config file and to reload Nginx.

Ingress IP address determination

Another interesting aspect is the subsystem that updates the Ingress object's "Status.Address" field, which is supposed to contain the public IP address on which this Ingress is available. I wanted to know how that IP address is determined. Maybe the ingress controller performs some special magic to allocate a new IP address from the cloud provider?

It was not immediately obvious where the "Status.Address" field is updated. But browsing through the Kubernetes dashboard, I found that the nginx-ingress-controller pod printed the following message:

```
I0326 08:01:49.534479       5 status.go:352] updating Ingress
default/hello-ingress status to [{172.16.103.162 }]
```

So I grepped for the string "updating Ingress" in the kubernetes/ingress-nginx source directory, and found internal/ingress/status/status.go. This file implements the `StatusSyncher` class. In instance of this class is created in the NGINXController constructor. It spawns a background goroutine which, once a minute, queries the IP address of the node on which the Nginx ingress controller is running, and simply updates the "Status.Address" to that value.

When multiple Nginx ingress controllers are running (e.g. if you scaled the associated deployment to a number > 1), then these controllers run a leader election algorithm to ensure that only one controller instance updates "Status.Address".

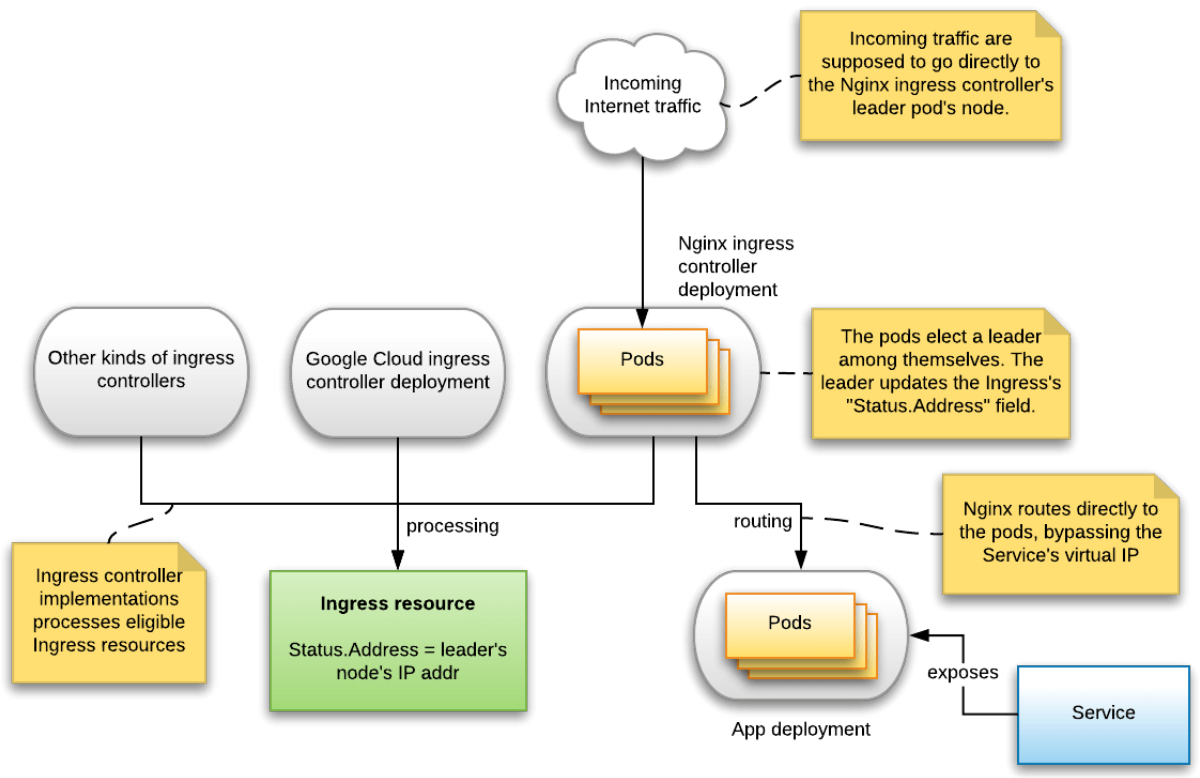
I'm kinda disappointed. No IP allocation magic from the cloud provider. It's just reporting its own IP address.

Not using Services

According to the Nginx ingress controller's documentation, it does not route traffic to the associated Service's virtual IP address. Instead it routes directly to the pods' IP addresses, using the endpoints API, for performance reasons and to allow features like session affinity:

> "[It routes to pods directly] in order to bypass kube-proxy to allow NGINX features like session affinity and custom load balancing algorithms. It also removes some overhead, such as conntrack entries for iptables DNAT."

Conclusion



The Nginx ingress controller turns out to be less magical than I first thought.

Ingress controllers

Ingress resources don't do anything by themselves: they are processed by **ingress controllers**, which vanilla Kubernetes does not provide by default. Managed Kubernetes providers may pre-install an appropriate ingress controller for you, e.g. Google Kubernetes Engine pre-installs a GCE ingress controller which provisions Google Cloud load balancers.

There is a community-developed Nginx ingress controller which provisions an Nginx instance to handle Ingress resources. It is deployed via a Deployment of the `quay.io/kubernetes-ingress-controller/nginx-ingress-controller:0.9.0` image.

When multiple ingress controllers are installed, one may run into the caveat that multiple ingress controllers try to handle the Ingress resources. This can be prevented by setting the `kubernetes.io/ingress.class` which instructs only specific controllers to process this Ingress.

How the Nginx ingress controller works

The Nginx ingress controller runs an Nginx installation inside itself. Thus, an Nginx ingress controller *is* an Nginx installation: it does not provision additional Nginx pods or something.

The Nginx ingress controller works by listening for create/read/update events on resources it is interested in, namely Ingresses, Endpoints, Secrets and ConfigMaps. Every time such an event is detected, it reconfigures the internal Nginx to do the right thing by writing out an appropriate Nginx config file and then telling Nginx to reload.

The Nginx installations route directly to the pods' IP addresses, bypassing the associated Service's virtual IP. This increases performance and allows Nginx features such as session affinity.

Ingress IP address publication

The Nginx ingress controller merely sets the Ingress resource's "Status.Address" field to its own node's IP address. There is no special IP address provisioning magic going on.

If you scale the Nginx ingress controller's deployment to more than 1, then each instance will run a leader election algorithm so that only 1 controller instance gets to set the "Status.Address" field.

Disappointed & relieved at the same time

On the one hand I'm disappointed that there are no more interesting things, on the other hand I'm relieved because "more interesting" means "more complex".

Hi, I'm Hongli Lai

I'm a software developer, CTO, business consultant and entrepreneur. I mentor people to help them grow. I write my software with the belief that software development and work should be fun, and that computers should serve humans instead of the other way around.

About Hongli

Joyful Bikeshedding

subscribe by RSS

or check me out on social media