Configuring HTTPS to a Web Service on Google Kubernetes Engine

Upgrade
S Fong
Follow
Mar 8, 2018 · 11 min read

Once again, this was harder than expected on Kubernetes as the most obvious tutorials on exposing cluster services to the outside world stop at HTTP. The documentation suggests that HTTPS can be configured but details were scant and scattered all over.

Exposing the Deployment

Let's start with something simple. Exposing the deployment to the Internet so we can see our new web service. This is from the Step 5 of the Hello App Tutorial. I'm using Coursemology again, but you can use your own service or the container in the Hello App tutorial.

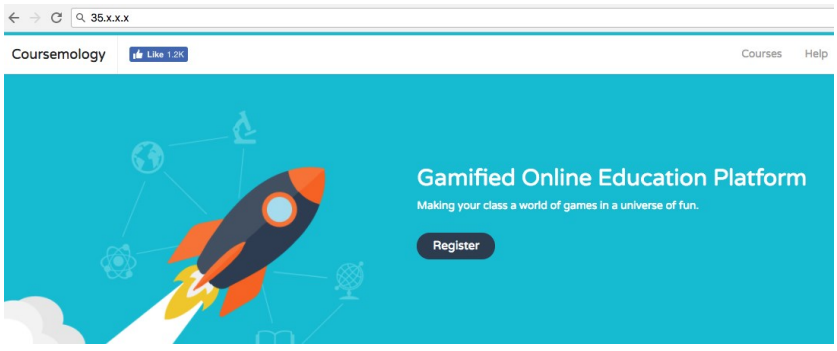
```
kubectl expose deployment DEPLOYMENT --type=LoadBalancer
```

The key thing here is to specify the `type` and your deployment's name. This will expose your deployment through a service and set up a Load Balancer on Google Cloud Platform.

Check out your new service. You can see the external IP here too:

```
$ kubectl get services
NAME         TYPE          CLUSTER-IP    EXTERNAL-IP PORT(S)
AGE
DEPLOYMENT   LoadBalancer 10.x.x.x      35.x.x.x     80:30853/TCP
6m
```

Visit the external IP in a web browser and view your app.



Exposing a deployment

Note that there is no HTTPS. In the GCP web console, select Network

Services -> Load Balancing from the menu, and observe that a Load Balancer has been automatically created. As the tutorial warns, this is subject to billing.

Clean up by deleting the service:

```
$ kubectl delete service DEPLOYMENT
```

Refresh the Load Balancing page in the GCP web console to verify that the load balancer has been removed.

Basic Ingress

A Kubernetes Ingress is a set of rules that allows inbound traffic to reach the cluster's services.

I only have my app server to expose, so let's use the example for a single service ingress:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: basic
spec:
  backend:
    serviceName: SERVICE
    servicePort: 80
```

Create the ingress on the cluster with `kubectl create -f ingress.yml`. Like the other Kubernetes resources, you can see what is going on with `kubectl get ingress`:

```
$ kubectl get ingress
NAME      HOSTS     ADDRESS         PORTS     AGE
basic     *         35.201.74.164   80        1h
```

Wait a while for the Load Balancer and forwarding rules to be created, then visit the page. Oops, something is missing.

Where's my web service?

The Ingress specified a Service, but we have not created one. In the previous example, exposing the deployment automatically created the Service, but the Ingress does not do this. Let's create the Service:

```
apiVersion: v1
kind: Service
metadata:
  name: cm-web
spec:
  selector:
    app: web
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

The important thing here is specifying the type of the Service as `NodePort`. This allocates a high port on each node in the cluster which will proxy requests to the Service. In this example, it is port 30598.

```
$ kubectl get services
NAME    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
cm-web  NodePort    10.x.x.x        <none>        80:30598/TCP
4m
```

Visit the Ingress' external IP and it should work. I say 'should' because Coursemology had some other configuration and data that I had to edit, and I ended up recreating the Ingress before everything worked. A simpler Service should just work when it comes up.

Google's Load Balancer performs health checks on the associated backend service. The service must return a status of 200. If it does not, the load balancer marks the instance as unhealthy and does not send it any traffic until the health check shows that it is healthy again. This caused some complications for Coursemology, which will be explained later.

Ingress with TLS

Time to move on to securing the connection with an SSL certificate. The TLS Section of the Ingress documentation explains that it is simply a matter of including the certificates in a Kubernetes Secret, then specifying the certificate in the Ingress. You will not be able to use this until you have SSL certificates, but here is the updated Ingress file:

```
apiVersion: v1
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded private key
kind: Secret
metadata:
  name: sslcerts
  namespace: default
type: Opaque
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: tls-ingress
spec:
  tls:
  - secretName: sslcerts
  backend:
    serviceName: cm-web
    servicePort: 80
```

Ingress Controllers

Let's pause here to talk about the concept of Ingress Controllers. This was confusing for me as GKE provides one by default and so their tutorials do not mention it. However, other posts frequently talk about needing one and it is mentioned in the documentation.

As mentioned in the explanation of the concepts, the Ingress resource will not work without an Ingress controller. GCE and Nginx are the two supported Ingress controllers. They are not the only options. Traefik is another popular one and is used by my colleagues on our self-hosted Kubernetes cluster.

By default, GCP uses the GCE Ingress Controller. You can run multiple Ingress controllers and select which controller should handle an Ingress resource by adding an annotation to the Ingress metadata. See this section in the Nginx Ingress controller documentation for an example. With this in mind, the section in this post explaining how to use Traefik on Kubernetes finally made sense to me.

Using a Static External IP

A domain name is needed for an SSL certificate. We also want to create a fixed 'A record' for it on the name registrar. With an Ingress, the external IP keeps changing as it is deleted and created. We can solve this problem on GCP by reserving an external IP address which we can then assign to the Ingress each time.

To reserve an external IP, follow the instructions here. There are two kinds of IP addresses, create a global one. I used the web console for this as I only needed one address, which I would then be keeping for some time. Note that unused reserved IP addresses will cost money. They are free if they are assigned to something.

Now edit your Ingress to use this IP address. Kelsey Hightower has a tutorial. The main thing is to add an annotation `kubernetes.io/ingress.global-static-ip-name` to the Ingress metadata. Official documentation can be found here, at Step 2b.

The Ingress section should now look something like this:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: tls-ingress
  annotations:
    kubernetes.io/ingress.global-static-ip-name: "my-ip-name"
spec:
  tls:
  - secretName: sslcerts
  backend:
    serviceName: cm-web
    servicePort: 80
```

Obtaining an SSL Certificate

Let's Encrypt provides free SSL certificates. However, the certificates have a short validity of three months. Certbot automates the process of renewing and replacing the certificates on conventional web servers. For Kubernetes, there are some solutions and blog posts. Unfortunately these are either a little dated or not production ready. Here are links to some of those solutions:

1. Runnable.com
2. Ployst Blog, Vadosware Blog
3. Kube Lego (no longer maintained)
4. Kube Lego's successor Cert Manager (not production ready)

The general approach is to run a Service inside the Kubernetes cluster which will renew the certificates and update the Kubernetes Secret holding the certificates. From the Runnable.com blog post, it looks like changing

the `label` key on the Ingress will then cause Kubernetes to reload the secrets, which should populate the new certificates up to the Google Load Balancer.

'Should' is the word used here again because I have yet to try this. I decided to kick the proverbial can of automating certificate renewal down the road.

Instead, I decided to just install `certbot` locally and generate something I can use first. On OS X, just run `brew install certbot` . Then let's try running it:

```
$ certbot
Traceback (most recent call last):
  File "/usr/local/bin/certbot", line 6, in <module>
    from pkg_resources import load_entry_point
...
  raise DistributionNotFound(req, requirers)
pkg_resources.DistributionNotFound: The 'certbot==0.21.1'
distribution was not found and is required by the application
```

That does not look right. I couldn't find anything on Google, but it finally occured to me that it seemed to be trying to load a package which could not be found, so I tried `pip3 install certbot` . After that, `certbot` could run! I am guessing that the problem comes from Homebrew's Python and the system installation of Python conflicting with each other and messing up the module search paths for other programs.

`certbot` can be told to use DNS verification and to only generate certificates, without trying to install them onto a web server. It usually needs `sudo` to run, but that can be avoided by specifying different directories for it to write to. Let's put this all together to generate a certificate for the domain:

```
$ certbot -d DOMAIN_NAME --manual --logs-dir certbot --config-dir
certbot --work-dir certbot --preferred-challenges dns certonly
```

Follow the prompts. You must allow it to log your IP address or you cannot use the service.

When you reach the following prompt, go to your registrar's webpage and add a DNS TXT record:

```
-------------------------------------------------------------------
--
Please deploy a DNS TXT record under the name
_acme-challenge.DOMAIN with the following value:

LONG_RANDOM_CHALLENGE_STRING

Before continuing, verify the record is deployed.
-------------------------------------------------------------------
--
Press Enter to Continue
```

Check that the record has populated with the command `dig @8.8.8.8 TXT _acme-challenge.DOMAIN` , remembering to replace `DOMAIN` with your own domain name. Once you see the challenge string in the `ANSWER_SECTION`, you can press Enter to continue the certificate issuance process.

Since I specified that `certbot` should use a directory named `certbot` , the newly generated certificates are in the directory `certbot/live/DOMAIN` . The necessary files are `fullchain.pem` and `privkey.pem` , but their contents have to be base64 encoded first before adding them to the Secret defined in the Ingress file.

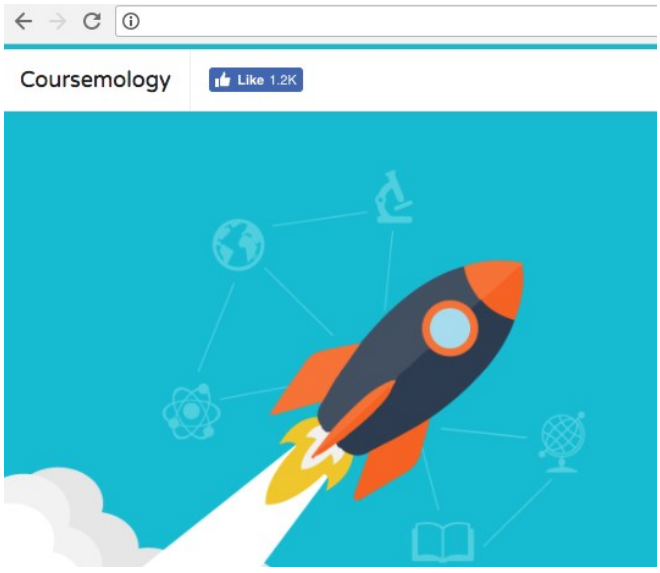
To encode the certificate and private key, run the following commands in the folder with the certificates:

```
$ cat privkey.pem | base64
LONGBASE64ENCODEDKEY
$ cat fullchain.pem | base64
LONGBASE64ENCODEDCERT
```

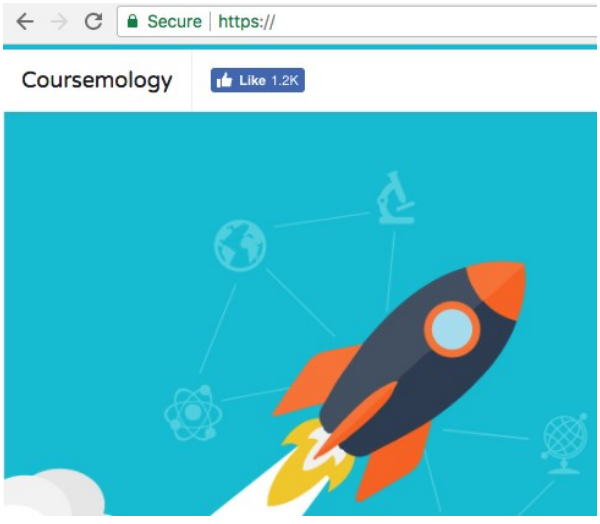
Copy `LONGBASE64ENCODEDKEY` into the `tls.key` value in the Ingress file, and `LONGBASE64ENCODEDCERT` into the `tls.crt` value in the Ingress file. If you prefer, you can define the `sslcerts` Secret in its own file.

Replace the basic Ingress by first deleting it with `kubectl delete ingress basic` , then create the TLS Ingress with your updated file using the command `kubectl create -f ingress.yml` .

You should now be able to access the service using both HTTP and HTTPS. The domain name has been removed in the screenshots below.



HTTP access

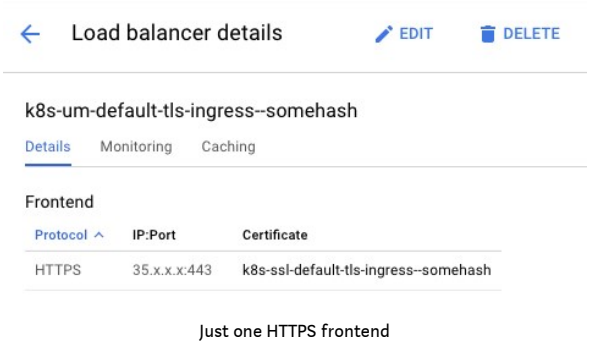


HTTPS access

Enforcing HTTPS

With the default GCE Ingress controller, one way of enforcing HTTPS is to simply not allow HTTP. This can be done quite easily be adding an annotation to the Ingress definition. See the Blocking HTTP section of the GCE Ingress documentation.

Let's add that to the YAML file and replace the Ingress. In the Load Balancer details on the GCP web console, notice that the frontend now only has a single entry for HTTPS.



Just one HTTPS frontend

Visiting the HTTP URL also gives a 404 from Google.

This works, but could lead to a poor user experience as users who just type the URL into the address bar might omit the HTTPS and think the website is down. Ideally, we should redirect users to the HTTPS site.

Redirect to HTTPS

Remove the annotation for HTTPS only from the Ingress file and recreate the Ingress.

GCE's Ingress controller does not support redirect rules, but the documentation includes a solution. The backend Service must inspect the `x-forwarded-proto` header and redirect if the value is equal to `http` . The Coursemology pod does have Nginx in front of it so I could add the following block to the Nginx configuration file:

```
if ($http_x_forwarded_proto = "http") {
    return 301 https://$host$request_uri;
}
```

The check **MUST** be for http. Although theoretically equivalent, if the check is written as `if($http_x_forwarded != 'https')` as suggested in this Serverfault question, Google Cloud Load Balancer's automated health checks will end up getting a 301 redirect and the backend will be considered unhealthy. Unhealthy backends get no traffic. This is mentioned right at the bottom of the section on redirecting:

> *Note that the GCLB health checks* do not *get the* `301` *because they don't include* `x-forwarded-proto`*.*

My Nginx configuration is stored in a config map, so to put it into effect, the config map had to be replaced. The pod then has to be deleted so the

Deployment will create a new one and pick up the new config map value.

Now the Load Balancer frontend shows both a HTTP and a HTTPS entry. Visiting the HTTP version of the website also causes a redirect to HTTPS.

Automating Let's Encrypt Certificate Renewal

There are a couple of ways to do this, but for now this can has been kicked down the road.

As mentioned previously, one way is to run a Let's Encrypt Service inside the Kubernetes cluster which will automatically renew the certificates and update the Secret. The Ingress then has to be reloaded to pick up the new certificate. A Kubernetes CronJob could be useful here.

Another method is to avoid Google Cloud Load Balancer entirely. Create a cheap little VM as a load balancer and run Traefik, which supports Let's Encrypt renewal out of the box. With this setup, it will also be possible to use Nginx and Certbot. Either way, point the load balancer to the NodePort on the internal IP addresses of the Kubernetes cluster's nodes. As long as the original set of nodes does not go down and get recreated with new IP addresses, no additional configuration will be needed. This solution is also cheaper, but a manually configured VM as a load balancer is probably more likely to fail than Google's Cloud Load Balancer solution.

I will probably move to the second solution in future. Google Cloud Load Balancer's forwarding rules cost 2.5 cents each per hour. This does not seem like much, but there is one rule for HTTP and another for HTTPS, which comes up to 5 cents an hour. Over one month, the bill for the forwarding rules comes to US$37.20 in the Iowa region, enough to run one n1-standard-1 Compute instance with money to spare.

. . .

Allowing the Health Check on Coursemology

Coursemology allows multi-tenancy. What this means is that accessing the site through different subdomains shows a different set of courses and users. Originally, going to an invalid subdomain simply caused the user to see the default tenant. However, user typos caused great confusion for users so subdomains which are not tenants now return a 404.

This sensible user friendly change caused some complications with Google Load Balancer's Health Check. Since the health check uses a host that is not a tenant, Coursemology returns a 404. The health check fails and then the backend service gets no traffic even though everything is working as designed.

Checking the logs in the Nginx container with the command `kubectl logs POD_NAME -c nginx` (hint: tab completion works for the pod names, you do not have to type the whole thing), the health check entries are quite obvious.

```
10.x.x.x - - [08/Mar/2018:07:14:29 +0000] "GET / HTTP/1.1" 404
1564 "-" "GoogleHC/1.0" "-"
10.x.x.x - - [08/Mar/2018:07:14:29 +0000] "GET / HTTP/1.1" 404
1564 "-" "GoogleHC/1.0" "-"
10.x.x.x - - [08/Mar/2018:07:14:29 +0000] "GET / HTTP/1.1" 404
1564 "-" "GoogleHC/1.0" "-"
```

They have a UserAgent value of `GoogleHC` ! By changing the Host header in Nginx, the Rails app could be tricked into thinking the request is for a valid tenant. This StackOverflow answer explains how to do it.

```
# Change host for Google HealthCheck so it'll hit the localhost
# instance and return 200.
set $my_host $http_host;
if ($http_user_agent ~* "^GoogleHC") {
  set $my_host "localhost";
}

...

location @app {
  proxy_pass http://app;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header Host $my_host;
  proxy_set_header X-Forwarded-Proto https;
  proxy_redirect off;
}
```

In Coursemology's database, I added a fake tenant for "localhost". This is not a new hack, we are already doing this so our current deployment pipeline can ensure that Puma starts before reporting a successful deployment.

Conclusion

Moving forward, TLS is going to be a minimum requirement for every website. Google Chrome will soon be more aggressive at marking HTTP websites as not secure. Encrypting the connection is even more important for sites that require login and request usernames and passwords.

Hopefully, this post has helped you understand how to setup HTTPS for your own site on Kubernetes too.

Kubernetes | Google Cloud Platform | Ssl | Lets Encrypt

WRITTEN BY
LH Fong
Follow

Engineering Tomorrow's Systems
Follow
ESTL is a leading full-stack engineering shop in Singapore's Ministry of Education. Our engineers build cutting-edge software to simplify the everyday duties of teachers and admin staff alike.

See responses (7)

More From Medium

Related reads

Why HPA failed to work on GKE, and how we fixed it

Jiang Huan in Titansoft Engineering Blog
Oct 11, 2019 · 5 min read ★

185

Related reads

How to Deploy MongoDB on Google Kubernetes Engine (GKE)

Grigor Khachatryan in devgorilla
Mar 11, 2019 · 4 min read ★

402

Related reads

Create ReadWriteMany PersistentVolumeClaims on your Kubernetes Cluster

Lexa in ASL19 Developers
May 17, 2019 · 8 min read ★

229