**Automated SSL Termination in EKS with Cert Manager(Lets-Encrypt)**

Jaswanth Kumar Jonnalagadda

Amazon Web Services

October 2019



© 2019 Amazon Web Services, Inc. or its affiliates. All rights reserved. This work may not be reproduced or redistributed, in whole or in part, without

prior written permission from Amazon Web Services, Inc. Commercial copying, lending, or selling is prohibited.  
All trademarks are the property of their owner

Table of Contents

[Kubernetes Cert-Manager: 2](#_Toc22557454)

[Pre-Requisite: 3](#_Toc22557455)

[Installing Cert-Manager: 3](#_Toc22557456)

[Verifying Functionality: 4](#_Toc22557457)

[Configuring Issuer: 5](#_Toc22557458)

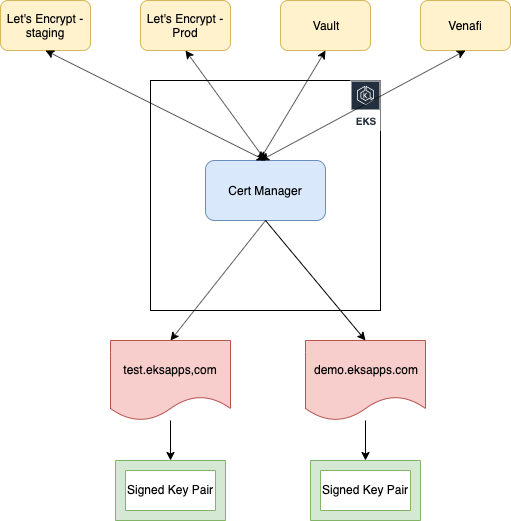
[Configuring with Let’s Encrypt Cluster-Issuer: 5](#_Toc22557459)

# **Kubernetes Cert-Manager**:

cert-manager is a native Kubernetes certificate management controller.

cert-manager can help with issuing certificates from a variety of sources, such as Let’s Encrypt, HashiCorp Vault, Venafi, a simple signing keypair, or self-signed. cert-manager will ensure certificates are valid and up to date, and attempt to renew certificates at a configured time before expiry.

Architecture:



# **Pre-Requisite**:

Following are the expected perquisites for the cert manager to issue certificate dynamically.

1. Fully functional EKS cluster.
2. Domain of your own.

# **Installing Cert-Manager**:

Follow the below steps to install cert-manager in the EKS cluster.

1. Create a separate namespace in the EKS cluster for cert-manager
   * **kubectl create namespace cert-manager**

cat <<EOF > sample-resources.yaml

apiVersion: v1

kind: Namespace

metadata:

name: cert-manager-check

---

apiVersion: cert-manager.io/v1alpha2

kind: Issuer

metadata:

name: test-selfsigned

namespace: cert-manager-test

spec:

selfSigned: {}

---

apiVersion: cert-manager.io/v1alpha2

kind: Certificate

metadata:

name: selfsigned-cert

namespace: cert-manager-test

spec:

commonName: example.com

secretName: selfsigned-cert-tls

issuerRef:

name: test-selfsigned

EOF

1. Install cert manager in the namespace with the below command.
   * **kubectl apply -f** [**https://github.com/jetstack/cert-manager/releases/download/v0.11.0/cert-manager.yaml**](https://github.com/jetstack/cert-manager/releases/download/v0.11.0/cert-manager.yaml) **--validate=false**

**\*\*** Reason for the –validate=false tag is to overcome the way kubectl performs resource validation on kubernetes version less that 1.15

1. Verify Installation with the below command
   * **kubectl get pods --namespace cert-manager**
   * If everything went well we will see three pods in running state as shown
   * To view the custom resources that are deployed along with cert-manager, run the following command: **kubectl get crd --all-namespaces**

## **Verifying Functionality:**

1. Build a sample self-signed certificate issuer in the cluster with the following script
2. Execute the script with the command
   * **Kubectl apply -f sample-resources.yaml**
3. Wait for few seconds for the cert-manager to process the certificate request.
   * Execute the command to describe the generated certificate
   * **Kubectl describe cert -n cert-manager-test**

You should see a similar output

With this we can confirm that the cert manager is installed without any errors.

Clean up test resources with the command: **kubectl delete -f sample-resources.yaml**

# **Configuring Issuer:**

Before you can begin issuing certificates, you must configure at least one Issuer or ClusterIssuer resource in your cluster.

These represent a certificate authority from which signed x509 certificates can be obtained, such as Let’s Encrypt, or your own signing key pair stored in a Kubernetes Secret resource. They are referenced by Certificate resources in order to request certificates from them.

An Issuer is scoped to a single namespace, and can only fulfill Certificate resources within its own namespace. This is useful in a multi-tenant environment where multiple teams or independent parties operate within a single cluster.

On the other hand, a ClusterIssuer is a cluster wide version of an Issuer. It is able to be referenced by Certificate resources in any namespace.

In this article we will use Let’s Encrypt cluster-issuer to explain issuer for a cluster level scope

## **Configuring with Let’s Encrypt Cluster-Issuer**:

1. Deploy an ingress-nginx using an ELB to expose the service.

Run the following commands to deploy the ingress controller.

* kubectl apply -f <https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/static/mandatory.yaml>
* kubectl apply -f <https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/static/provider/aws/service-nlb.yaml>

\*\* It will take few minutes for the ingress controller to be up.

1. Verify the deployed service with the command: **kubectl get service -n ingress-nginx**

Sample Output:

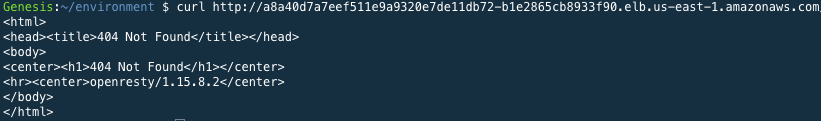


\*\* If the external-ip is not available, please wait for few minutes for the address to be issued.

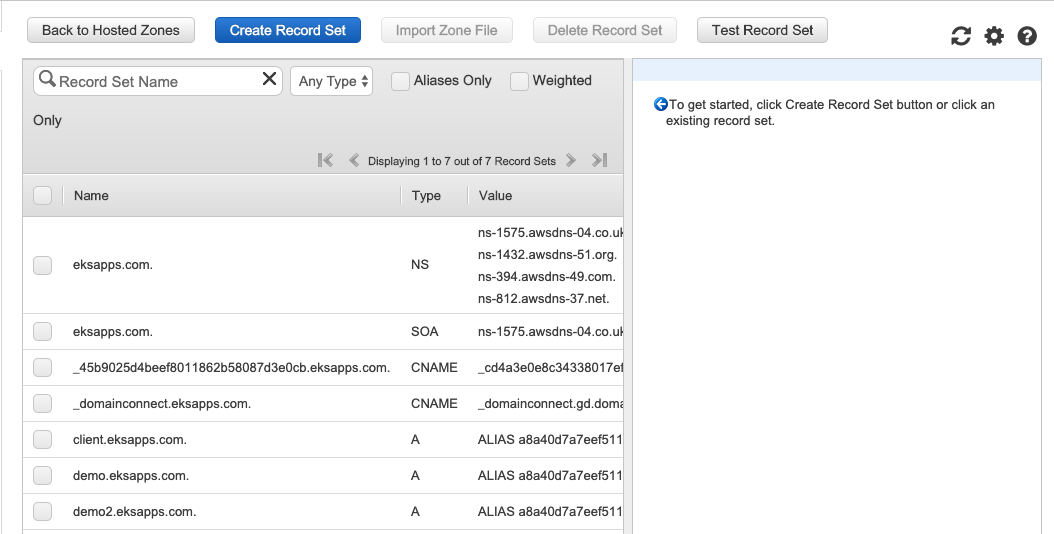
1. Once the external ip is issued, then verify if the traffic is being routed to the ingress-nginx

Command: **curl http:// a8a40d7a7eef511e9a9320e7de11db72-b1e2865cb8933f90.elb.us-east-1.amazonaws.com**

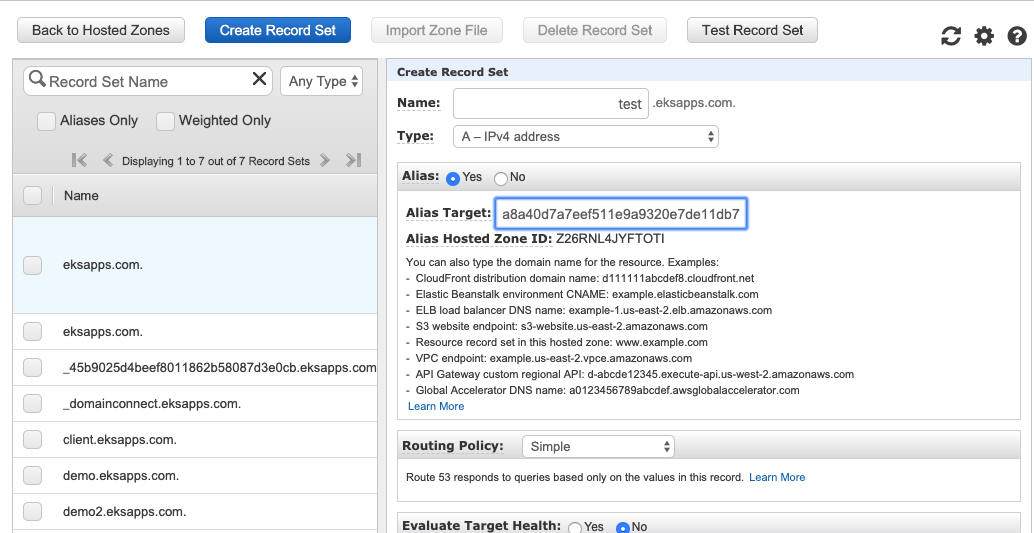
Sample Output:



1. Now that our NLB has been provisioned, we should point our application’s DNS records at the NLBs address. In the DNS provider’s console set an A record to pointing to your NLB external ip.



Click on create Record Set



Add the extern ip address

Choose a name

* Click on Create button.
* This will create a new entry in the DNS record set.
* This will resemble following

test.eksapps.com A [http://a8a40d7a7eef511e9a9320e7de11db72-b1e2865cb8933f90.elb.us-east-1.amazonaws.com](http://a8a40d7a7eef511e9a9320e7de11db72-b1e2865cb8933f90.elb.us-east-1.amazonaws.com/)

1. Create a namespace demo
   * Command: kubectl create namespace demo
2. Deploy a sample application in the demo namespace with the below deployment script

Command: **kubectl apply -f demo-application.yml -n demo**

---

apiVersion: v1

kind: Service

metadata:

name: appd

namespace: demo

spec:

type: ClusterIP

ports:

- port: 80

targetPort: 8080

selector:

app: hello-kubernetes

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: appd

namespace: demo

spec:

replicas: 2

selector:

matchLabels:

app: appd

template:

metadata:

labels:

app: appd

spec:

containers:

- name: appd

image: '682651395775.dkr.ecr.us-east-1.amazonaws.com/java\_app\_one:latest'

resources:

requests:

cpu: 100m

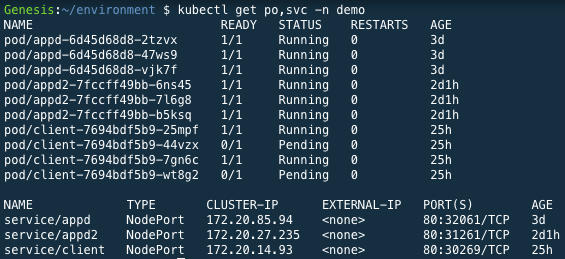
memory: 100Mi

ports:

- containerPort: 8080

1. Verify the application deployment with the below command
   * **Kubectl get po,svc -n demo**

Sample output:



1. Create a file letsencrypt-prod.yml and paste the script

apiVersion: cert-manager.io/v1alpha2

kind: ClusterIssuer

metadata:

name: letsencrypt-prod

spec:

acme:

# The ACME server URL

server: https://acme-v02.api.letsencrypt.org/directory

# Email address used for ACME registration

email: <email address>

# Name of a secret used to store the ACME account private key

privateKeySecretRef:

name: letsencrypt-prod

# Enable the HTTP-01 challenge provider

solvers:

- http01:

ingress:

class: nginx

Create this resource with the command: **kubectl apply -f letsencrypt-prod.yml**

1. To Verify the cluster issuer creation.
   * Command: **kubectl describe clusterissuer letsencrypt-prod**
   * Similar status should be observed with **Type: Ready**

. . .

Status:

Acme:

Last Registered Email: jaisai@gmail.com

Uri: https://acme-v02.api.letsencrypt.org/acme/acct/69491521

Conditions:

Last Transition Time: 2019-10-15T20:22:36Z

Message: The ACME account was registered with the ACME server

Reason: ACMEAccountRegistered

Status: True

Type: Ready

Events: <none>

1. As everything looks good, lets create an ingress for the deployed application.

Copy the below script to a file **ingress-tls-final.yml**

Domain we will be using

Secret we will be creating and using for this domain

This annotation shows that we are using cluster-issuer and the issuer is letsencrypt-prod

---

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: appd

namespace: demo

annotations:

kubernetes.io/ingress.class: "nginx"

cert-manager.io/cluster-issuer: letsencrypt-prod

labels:

app: appd

spec:

tls:

- hosts:

- demo.eksapps.com

secretName: letsencrypt-prod

rules:

- host: demo.eksapps.com

http:

paths:

- path: /

backend:

serviceName: appd

servicePort: 80

* + Create this ingress with the command: **kubectl apply -f ingress-tls-final.yml**
  + This will create a certificate and secret.
    1. That can be verified with the command: **kubectl get cert letsencrypt-prod -n demo**

Sample Status:

NAME READY SECRET AGE

letsencrypt-prod True letsencrypt-prod 5d19h

* + 1. Status of the cert can be seen with the command

**kubectl describe cert letsencrypt-prod -n demo**

Sample Status:

. . .

Spec:

Dns Names:

demo.eksapps.com

Issuer Ref:

Group: cert-manager.io

Kind: ClusterIssuer

Name: letsencrypt-prod

Secret Name: letsencrypt-prod

Status:

Conditions:

Last Transition Time: 2019-10-16T14:40:11Z

Message: Certificate is up to date and has not expired

Reason: Ready

Status: True

Type: Ready

Not After: 2020-01-14T13:40:10Z

Events: <none>

* + 1. Similar way we can see the status of the secret with the command

**kubectl describe secret letsencrypt-prod -n demo**

Sample Status:

Name: letsencrypt-prod

Namespace: demo

Labels: <none>

Annotations: cert-manager.io/alt-names: demo.eksapps.com,demo2.eksapps.com

cert-manager.io/certificate-name: letsencrypt-prod

cert-manager.io/common-name: demo.eksapps.com

cert-manager.io/ip-sans:

cert-manager.io/issuer-kind: ClusterIssuer

cert-manager.io/issuer-name: letsencrypt-prod

cert-manager.io/uri-sans:

Type: kubernetes.io/tls

Data

====

ca.crt: 0 bytes

tls.crt: 3586 bytes

tls.key: 1679 bytes

1. As the cert and secret are created, they are assigned to the ingress.

Now we can open the browser and reach the website over

**https://demo.eksapps.com**