cert-manager Documentation

Jetstack Ltd

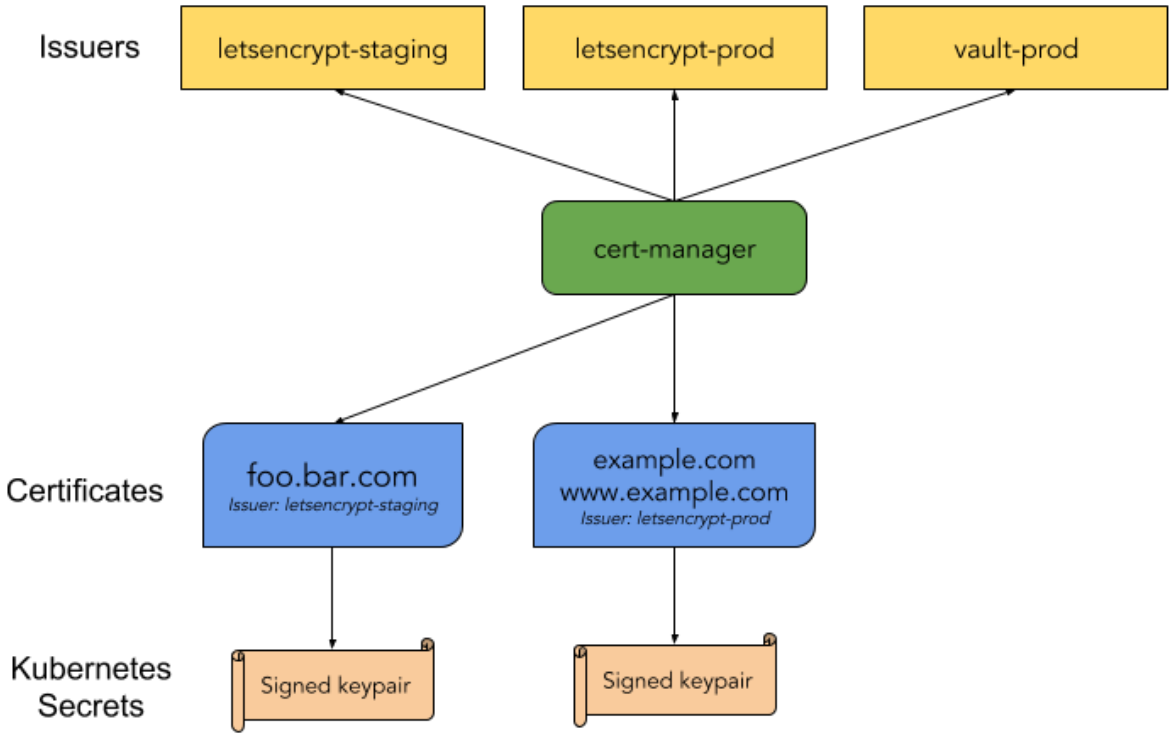
Nov 30, 2018

Contents:

cert-manager is a native Kubernetes certificate management controller. It can help with issuing certificates from a variety of sources, such as Let's Encrypt, HashiCorp Vault, a simple signing keypair, or self signed.

It will ensure certificates are valid and up to date, and attempt to renew certificates at a configured time before expiry.

It is loosely based upon the work of kube-lego and has borrowed some wisdom from other similar projects e.g. kube-cert-manager.



This is the full technical documentation for the project, and should be used as a source of references when seeking help with the project.

CHAPTER 1

Getting started

This contains information on getting started with cert-manager and deployment information.

New users should start here before proceeding to the *Tutorials* section, as the steps in this section are a prerequisite for all tutorials.

1.1 1. Configuring Helm and Tiller

Before deploying cert-manager, you must ensure Tiller is up and running in your cluster. Tiller is the server side component to Helm.

Your cluster administrator may have already setup and configured Helm for you, in which case you can skip this step.

Full documentation on installing Helm can be found in the Installing helm docs.

If your cluster has RBAC (Role Based Access Control) enabled (default in GKE v1.7+), you will need to take special care when deploying Tiller, to ensure Tiller has permission to create resources as a cluster administrator. More information on deploying Helm with RBAC can be found in the Helm RBAC docs.

1.2 2. Installing cert-manager

1.2.1 With Helm

Using Helm is the recommended way to deploy cert-manager. We publish a stable version of the chart to the public charts repository.

You can install the chart with the following command:

```
$ helm install \
    --name cert-manager \
    --namespace kube-system \
    stable/cert-manager
```

The default cert-manager configuration is good for the majority of users, but a full list of the available options can be found in the Helm chart README.

Note: If your cluster does not use RBAC (Role Based Access Control), you will need to disable creation of RBAC resources by adding `--set rbac.create=false` to your `helm install` command above.

Note: If you are upgrading from a previous release, please check the upgrading guide for special considerations.

1.2.2 With static manifests

As some users may not be able to run Tiller in their own environment, static Kubernetes deployment manifests are provided which can be used to install cert-manager.

You can get a copy of the static manifests from the deploy directory.

1.3 3. Configuring your first Issuer or ClusterIssuer

Before you can issue any Certificates, you will need to configure an *Issuer* or *ClusterIssuer* resource.

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.

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. Users often create `letsencrypt-staging` and `letsencrypt-prod` *ClusterIssuers* if they operate a single-tenant environment and want to expose a cluster-wide mechanism for obtaining TLS certificates from Let's Encrypt.

CHAPTER 2

Tutorials

This section contains numerous tutorials that cover basic use cases of cert-manager.

2.1 Quick-Start using Cert-Manager with NGINX Ingress

2.1.1 Step 0 - Install Helm Client

Skip this section if you have helm installed.

The easiest way to install *cert-manager* is to use Helm, a templating and deployment tool for Kubernetes resources.

First, ensure the Helm client is installed following the Helm installation instructions.

For example, on macOS:

```
$ brew install kubernetes-helm
```

2.1.2 Step 1 - Installer Tiller

Skip this section if you have Tiller set-up.

Tiller is Helm's server-side component, which the `helm` client uses to deploy resources.

Deploying resources is a privileged operation; in the general case requiring arbitrary privileges. With this example, we give Tiller complete control of the cluster. View the documentation on securing helm for details on setting up appropriate permissions for your environment.

Create the a ServiceAccount for tiller:

```
$ kubectl create serviceaccount tiller --namespace=kube-system
serviceaccount "tiller" created
```

Grant the `tiller` service account cluster admin privileges:

```
$ kubectl create clusterrolebinding tiller-admin --serviceaccount=kube-system:tiller -
↪-clusterrole=cluster-admin
clusterrolebinding.rbac.authorization.k8s.io "tiller-admin" created
```

Install tiller with the `tiller` service account:

```
$ helm init --service-account=tiller
$HELM_HOME has been configured at /Users/myaccount/.helm.

Tiller (the Helm server-side component) has been installed into your Kubernetes␣
↪Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow unauthenticated␣
↪users' policy.
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation see: https://docs.helm.sh/using_
↪helm/#securing-your-helm-installation
Happy Helming!
```

Update the helm repository with the latest charts:

```
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "stable" chart repository
...Successfully got an update from the "coreos" chart repository
Update Complete.  Happy Helming!
```

2.1.3 Step 2 - Deploy the NGINX Ingress Controller

A kubernetes ingress controller is designed to be the access point for HTTP and HTTPS traffic to the software running within your cluster. The nginx-ingress controller does this by providing an HTTP proxy service supported by your cloud provider's load balancer.

You can get more details about nginx-ingress and how it works from the documentation for nginx-ingress.

Use `helm` to install an Nginx Ingress controller:

```
$ helm install stable/nginx-ingress --name quickstart

NAME:   quickstart
LAST DEPLOYED: Sat Nov 10 10:25:06 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME                                  AGE
quickstart-nginx-ingress-controller  0s

==> v1beta1/ClusterRole
quickstart-nginx-ingress  0s

==> v1beta1/Deployment
quickstart-nginx-ingress-controller       0s
quickstart-nginx-ingress-default-backend  0s
```

(continues on next page)

(continued from previous page)

```
==> v1/Pod(related)

NAME                                                     READY   STATUS              ␣
↪RESTARTS  AGE
quickstart-nginx-ingress-controller-6cfc45747-wcxrg      0/1     ContainerCreating   0␣
↪         0s
quickstart-nginx-ingress-default-backend-bf9db5c67-dkg4l  0/1     ContainerCreating   0␣
↪         0s

==> v1/ServiceAccount

NAME                        AGE
quickstart-nginx-ingress    0s

==> v1beta1/ClusterRoleBinding
quickstart-nginx-ingress    0s

==> v1beta1/Role
quickstart-nginx-ingress    0s

==> v1beta1/RoleBinding
quickstart-nginx-ingress    0s

==> v1/Service
quickstart-nginx-ingress-controller       0s
quickstart-nginx-ingress-default-backend  0s


NOTES:
The nginx-ingress controller has been installed.
It may take a few minutes for the LoadBalancer IP to be available.
You can watch the status by running 'kubectl --namespace default get services -o wide␣
↪-w quickstart-nginx-ingress-controller'

An example Ingress that makes use of the controller:

  apiVersion: extensions/v1beta1
  kind: Ingress
  metadata:
    annotations:
      kubernetes.io/ingress.class: nginx
    name: example
    namespace: foo
  spec:
    rules:
      - host: www.example.com
        http:
          paths:
            - backend:
                serviceName: exampleService
                servicePort: 80
              path: /
    # This section is only required if TLS is to be enabled for the Ingress
    tls:
        - hosts:
            - www.example.com
```

(continues on next page)

(continued from previous page)

```
          secretName: example-tls

If TLS is enabled for the Ingress, a Secret containing the certificate and key must␣
↪also be provided:

  apiVersion: v1
  kind: Secret
  metadata:
    name: example-tls
    namespace: foo
  data:
    tls.crt: <base64 encoded cert>
    tls.key: <base64 encoded key>
  type: kubernetes.io/tls
```

It can take a minute or two for the cloud provider to provide and link a public IP address. When it is complete, you can see the external IP address using the `kubectl` command:

```
$ kubectl get svc

NAME                                        TYPE           CLUSTER-IP      EXTERNAL-IP␣
↪      PORT(S)                      AGE
kubernetes                                  ClusterIP      10.63.240.1     <none>      ␣
↪      443/TCP                      23m
quickstart-nginx-ingress-controller         LoadBalancer   10.63.248.177   35.233.154.
↪161    80:31345/TCP,443:31376/TCP   16m
quickstart-nginx-ingress-default-backend    ClusterIP      10.63.250.234   <none>      ␣
↪      80/TCP                       16m
```

This command shows you all the services in your cluster (in the `default` namespace), and any external IP addresses they have. When you first create the controller, your cloud provider won't have assigned and allocated an IP address through the LoadBalancer yet. Until it does, the external IP address for the service will be listed as `<pending>`.

Your cloud provider may have options for reserving an IP address prior to creating the ingress controller and using that IP address rather than assigning an IP address from a pool. Read through the documentation from your cloud provider on how to arrange that.

2.1.4 Step 3 - Assign a DNS name

The external IP that is allocated to the ingress-controller is the IP to which all incoming traffic should be routed. To enable this, add it to a DNS zone you control, for example as *example.your-domain.com*.

This quickstart assumes you know how to assign a DNS entry to an IP address and will do so.

2.1.5 Step 4 - Deploy an Example Service

Your service may have its own chart, or you may be deploying it directly with manifests. This quickstart uses manifests to create and expose a sample service. The example service uses kuard, a demo application which makes an excellent back-end for examples.

The quickstart example uses three manifests for the sample. The first two are a sample deployment and an associated service:

- deployment manifest: deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kuard
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: kuard
    spec:
      containers:
      - image: gcr.io/kuar-demo/kuard-amd64:1
        imagePullPolicy: Always
        name: kuard
        ports:
        - containerPort: 8080
```

- service manifest: service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kuard
spec:
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
  selector:
    app: kuard
```

You can create download and reference these files locally, or you can reference them from the GitHub source repository for this documentation. To install the example service from the tutorial files straight from GitHub, you may use the commands:

```
$ kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/master/
↪docs/tutorials/quick-start/example/deployment.yaml
deployment.extensions "kuard" created

$ kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/master/
↪docs/tutorials/quick-start/example/service.yaml
service "kuard" created
```

An ingress resource is what Kubernetes uses to expose this example service outside the cluster. You will need to download and modify the example manifest to reflect the domain that you own or control to complete this example.

A sample ingress you can start with is:

- ingress manifest: ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kuard
  annotations:
    kubernetes.io/ingress.class: "nginx"
    #certmanager.k8s.io/issuer: "letsencrypt-staging"
```

(continues on next page)

(continued from previous page)

```
    #certmanager.k8s.io/acme-challenge-type: http01

spec:
  tls:
  - hosts:
    - example.example.com
    secretName: quickstart-example-tls
  rules:
  - host: example.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: kuard
          servicePort: 80
```

You can download the sample manifest from github, edit it, and submit the manifest to Kubernetes with the command:

```
$ kubectl create --edit -f https://raw.githubusercontent.com/jetstack/cert-manager/
↪master/docs/tutorials/quick-start/example/ingress.yaml

# edit the file in your editor, and once it is saved:
ingress.extensions "kuard" created
```

Note: The ingress example we show above has a *host* definition within it. The nginx-ingress-controller will route traffic when the hostname requested matches the definition in the ingress. You *can* deploy an ingress without a *host* definition in the rule, but that pattern isn't usable with a TLS certificate, which expects a fully qualified domain name.

Once it is deployed, you can use the command *kubectl get ingress* to see the status of the ingress:

```
NAME      HOSTS     ADDRESS   PORTS     AGE
kuard     *                   80, 443   17s
```

It may take a few minutes, depending on your service provider, for the ingress to be fully created. When it has been created and linked into place, the ingress will show an address as well:

```
NAME      HOSTS     ADDRESS          PORTS     AGE
kuard     *         35.199.170.62    80        9m
```

Note: The IP address on the ingress *may not* match the IP address that the nginx-ingress-controller. This is fine, and is a quirk/implementation detail of the service provider hosting your Kubernetes cluster. Since we are using the nginx-ingress-controller instead of any cloud-provider specific ingress backend, use the IP address that was defined and allocated for the nginx-ingress-service LoadBalancer resource as the primary access point for your service.

Make sure the service is reachable at the domain name you added above, for example *http://example.your-domain.com*. The simplest way is to open a browser and enter the name that you set up in DNS, and for which we just added the ingress.

You may also use a command line tool like *curl* to check the ingress.

```
$ curl -kivL -H 'Host: example.your-domain.com' 'http://35.199.164.14'
```

The options on this curl command will provide verbose output, following any redirects, show the TLS headers in

the output, and not error on insecure certificates. With nginx-ingress-controller, the service will be available with a TLS certificate, but it will be using a self-signed certificate provided as a default from the nginx-ingress-controller. Browsers will show a warning that this is an invalid certificate. This is expected and normal, as we have not yet used cert-manager to get a fully trusted certificate for our site.

> **Warning:** It is critical to make sure that your ingress is available and responding correctly on the internet. This quickstart example uses Let's Encypt to provide the certificates, which expects and validates both that the service is available and that during the process of issuing a certificate uses that valdiation as proof that the request for the domain belongs to someone with sufficient control over the domain.

2.1.6 Step 5 - Deploy Cert Manager

We need to install cert-manager to do the work with kubernetes to request a certificate and respond to the challenge to validate it. We can use helm to install cert-manager. This example installed cert-manager into the *kube-system* namespace from the public helm charts.

```
$ helm install --name cert-manager --namespace cert-manager stable/cert-manager

NAME:   cert-manager
LAST DEPLOYED: Sat Nov 17 09:09:02 2018
NAMESPACE: cert-manager
STATUS: DEPLOYED

RESOURCES:
==> v1/Pod(related)
NAME                           READY  STATUS              RESTARTS  AGE
cert-manager-6f9ffcc9cc-rfwn5  0/1    ContainerCreating   0         0s

==> v1/ServiceAccount

NAME          AGE
cert-manager  0s

==> v1beta1/ClusterRole
cert-manager  0s

==> v1beta1/ClusterRoleBinding
cert-manager  0s

==> v1beta1/Deployment
cert-manager  0s


NOTES:
cert-manager has been deployed successfully!

In order to begin issuing certificates, you will need to set up a ClusterIssuer
or Issuer resource (for example, by creating a 'letsencrypt-staging' issuer).

More information on the different types of issuers and how to configure them
can be found in our documentation:

https://cert-manager.readthedocs.io/en/latest/reference/issuers.html
```

(continues on next page)

(continued from previous page)

```
For information on how to configure cert-manager to automatically provision
Certificates for Ingress resources, take a look at the `ingress-shim`
documentation:

https://cert-manager.readthedocs.io/en/latest/reference/ingress-shim.html
```

Cert-manager uses two different custom resources, also known as CRD's, to configure and control how it operates, as well as share status of its operation. These two resources are:

Issuers (or *ClusterIssuers*)

An Issuer is the definition for where cert-manager will get request TLS certificates. An Issuer is specific to a single namespace in Kubernetes, and a ClusterIssuer is meant to be a cluster-wide definition for the same purpose.

Certificate

A certificate is the resource that cert-manager uses to expose the state of a request as well as track upcoming expirations.

2.1.7 Step 6 - Configure Let's Encrypt Issuer

We will set up two issuers for Let's Encrypt in this example. The Let's Encrypt production issuer has very strict rate limits. When you are experimenting and learning, it is very easy to hit those limits, and confuse rate limiting with errors in configuration or operation.

Because of this, we will start with the Let's Encrypt staging issuer, and once that is working switch to a production issuer.

Create this definition locally and update the email address to your own. This email required by Let's Encryppt and used to notify you of certificate expirations and updates.

```
  apiVersion: certmanager.k8s.io/v1alpha1
  kind: Issuer
  metadata:
    name: letsencrypt-staging
  spec:
    acme:
      # The ACME server URL
      server: https://acme-staging-v02.api.letsencrypt.org/directory
      # Email address used for ACME registration
      email: user@example.com
      # Name of a secret used to store the ACME account private key
      privateKeySecretRef:
        name: letsencrypt-staging
      # Enable the HTTP-01 challenge provider
      http01: {}
```

Once edited, apply the custom resource:

```
$ kubectl apply -f staging-issuer.yaml
issuer.certmanager.k8s.io "letsencrypt-staging" created
```

Also create a production issuer and deploy it. As with the staging issuer, you will need to update this example and add in your own email address.

```
  apiVersion: certmanager.k8s.io/v1alpha1
  kind: Issuer
  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: user@example.com
      # Name of a secret used to store the ACME account private key
      privateKeySecretRef:
        name: letsencrypt-prod
      # Enable the HTTP-01 challenge provider
      http01: {}
```

```
$ kubectl apply -f production-issuer.yaml
issuer.certmanager.k8s.io "letsencrypt-prod" created
```

Both of these issuers are configured to use the *HTTP01* challenge provider.

Check on the status of the issuer after you create it:

You should see the issuer listed with a registered account.

2.1.8 Step 7 - Deploy a TLS Ingress Resource

With all the pre-requisite configuration in place, we can now do the pieces to request the TLS certificate. There are two primary ways to do this: using annotations on the ingress with *ingress-shim* or directly creating a certificate resource.

In this example, we will add annotations to the ingress, and take advantage of ingress-shim to have it create the certificate resource on our behalf. After creating a certificate, the cert-manager will update or create a ingress resource and use that to validate the domain. Once verified and issued, cert-manager will create or update the secret defined in the certificate.

Note: The secret that is used in the ingress should match the secret defined in the certificate. There isn't any explicit checking, so a typo will resut in the nginx-ingress-controller falling back to its self-signed certificate. In our example, we are using annotations on the ingress (and ingress-shim) which will create the correct secrets on your behalf.

Edit the ingress add the annotations that were commented out in our earlier example:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kuard
  annotations:
    kubernetes.io/ingress.class: "nginx"
    certmanager.k8s.io/issuer: "letsencrypt-staging"
    certmanager.k8s.io/acme-challenge-type: http01

spec:
  tls:
  - hosts:
    - example.example.com
    secretName: quickstart-example-tls
```

(continues on next page)

(continued from previous page)

```
  rules:
  - host: example.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: kuard
          servicePort: 80
```

and apply it:

```
$ kubectl apply -f ingress.yaml
ingress.extensions "kuard" configured
```

Cert-manager will read these annotations and use them to create a certificate, which you can request and see:

```
$ kubectl get certificate
NAME                      AGE
quickstart-example-tls   38s
```

Cert-manager reflects the state of the process for every request in the certificate object. You can view this information using the *kubectl describe* command:

```
 $ kubectl describe certificate quickstart-example-tls

 Name:         quickstart-example-tls
 Namespace:    default
 Labels:       <none>
 Annotations:  <none>
 API Version:  certmanager.k8s.io/v1alpha1
 Kind:         Certificate
 Metadata:
   Cluster Name:
   Creation Timestamp:  2018-11-17T17:58:37Z
   Generation:          0
   Owner References:
     API Version:           extensions/v1beta1
     Block Owner Deletion:  true
     Controller:            true
     Kind:                  Ingress
     Name:                  kuard
     UID:                   a3e9f935-ea87-11e8-82f8-42010a8a00b5
   Resource Version:        9295
   Self Link:               /apis/certmanager.k8s.io/v1alpha1/namespaces/default/
↪certificates/quickstart-example-tls
   UID:                     68d43400-ea92-11e8-82f8-42010a8a00b5
 Spec:
   Acme:
     Config:
       Domains:
         example.your-domain.com
       Http 01:
         Ingress:
         Ingress Class:  nginx
   Dns Names:
     example.your-domain.com
```

(continues on next page)

(continued from previous page)

```
  Issuer Ref:
    Kind:       Issuer
    Name:       letsencrypt-staging
  Secret Name:  quickstart-example-tls
Status:
  Acme:
    Order:
      URL:  https://acme-staging-v02.api.letsencrypt.org/acme/order/7374163/13665676
  Conditions:
    Last Transition Time:  2018-11-17T18:05:57Z
    Message:               Certificate issued successfully
    Reason:                CertIssued
    Status:                True
    Type:                  Ready
    Last Transition Time:  <nil>
    Message:               Order validated
    Reason:                OrderValidated
    Status:                False
    Type:                  ValidateFailed
Events:
  Type     Reason          Age                From          Message
  ----     ------          ----               ----          -------
  Normal   CreateOrder     9m                 cert-manager  Created new ACME order,␣
↪attempting validation...
  Normal   DomainVerified  8m                 cert-manager  Domain "example.your-
↪domain.com" verified with "http-01" validation
  Normal   IssueCert       8m                 cert-manager  Issuing certificate...
  Normal   CertObtained    7m                 cert-manager  Obtained certificate␣
↪from ACME server
  Normal   CertIssued      7m                 cert-manager  Certificate issued␣
↪Successfully
```

The events associated with this resource and listed at the bottom of the *describe* results show the state of the request. In the above example the certificate was validated and issued within a couple of minutes.

Once complete, cert-manager will have created a secret with the details of the certificate based on the secret used in the ingress resource. You can use the describe command as well to see some details:

```
$ kubectl describe secret quickstart-example-tls

Name:         quickstart-example-tls
Namespace:    default
Labels:       certmanager.k8s.io/certificate-name=quickstart-example-tls
Annotations:  certmanager.k8s.io/alt-names=example.your-domain.com
              certmanager.k8s.io/common-name=example.your-domain.com
              certmanager.k8s.io/issuer-kind=Issuer
              certmanager.k8s.io/issuer-name=letsencrypt-staging

Type:  kubernetes.io/tls

Data
====
tls.crt:  3566 bytes
tls.key:  1675 bytes
```

Now that we have confidence that everything is configured correctly, you can update the annotations in the ingress to specify the production issuer:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kuard
  annotations:
    kubernetes.io/ingress.class: "nginx"
    certmanager.k8s.io/issuer: "letsencrypt-prod"
    certmanager.k8s.io/acme-challenge-type: http01

spec:
  tls:
  - hosts:
    - example.example.com
    secretName: quickstart-example-tls
  rules:
  - host: example.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: kuard
          servicePort: 80
```

```
$ kubectl apply -f ingress.yaml

ingress.extensions "kuard" configured
```

You will also need to delete the existing secret, which cert-manager is watching and will cause it to reprocess the request with the updated issuer.

```
$ kubectl delete secret quickstart-example-tls

secret "quickstart-example-tls" deleted
```

This will start the process to get a new certificate, and using describe you can see the status. Once the production certificate has been updated, you should see the example KUARD running at your domain with a signed TLS certificate.

```
 $ kubectl describe certificate

 Name:         quickstart-example-tls
 Namespace:    default
 Labels:       <none>
 Annotations:  <none>
 API Version:  certmanager.k8s.io/v1alpha1
 Kind:         Certificate
 Metadata:
   Cluster Name:
   Creation Timestamp:  2018-11-17T18:36:48Z
   Generation:          0
   Owner References:
     API Version:           extensions/v1beta1
     Block Owner Deletion:  true
     Controller:            true
     Kind:                  Ingress
     Name:                  kuard
     UID:                   a3e9f935-ea87-11e8-82f8-42010a8a00b5
   Resource Version:        283686
```

(continues on next page)

(continued from previous page)

```
  Self Link:               /apis/certmanager.k8s.io/v1alpha1/namespaces/default/
↪certificates/quickstart-example-tls
  UID:                     bdd93b32-ea97-11e8-82f8-42010a8a00b5
Spec:
  Acme:
    Config:
      Domains:
        example.your-domain.com
      Http 01:
        Ingress:
        Ingress Class:  nginx
  Dns Names:
    example.your-domain.com
  Issuer Ref:
    Kind:       Issuer
    Name:       letsencrypt-prod
  Secret Name:  quickstart-example-tls
Status:
  Acme:
    Order:
      URL:  https://acme-v02.api.letsencrypt.org/acme/order/45980184/182533829
  Conditions:
    Last Transition Time:  2018-11-19T19:16:10Z
    Message:               Certificate issued successfully
    Reason:                CertIssued
    Status:                True
    Type:                  Ready
    Last Transition Time:  <nil>
    Message:               Order validated
    Reason:                OrderValidated
    Status:                False
    Type:                  ValidateFailed
Events:
  Type    Reason          Age   From          Message
  ----    ------          ----  ----          -------
  Normal  CreateOrder     26s   cert-manager  Created new ACME order, attempting␣
↪validation...
  Normal  DomainVerified  9s    cert-manager  Domain "example.your-domain.com"␣
↪verified with "http-01" validation
  Normal  IssueCert       8s    cert-manager  Issuing certificate...
  Normal  CertObtained    6s    cert-manager  Obtained certificate from ACME server
  Normal  CertIssued      6s    cert-manager  Certificate issued successfully
```

2.2 ACME Issuer Tutorials

This section contains tutorials that specifically utilise the ACME Issuer. They are designed to help teach the underlying concepts of cert-manager whilst also helping 'quickstart' common use-cases for the project.

2.2.1 Migrating from kube-lego

kube-lego is an older Jetstack project for obtaining TLS certificates from Let's Encrypt (or another ACME server).

Since cert-managers release, kube-lego has been gradually deprecated in favour of this project. There are a number of key differences between the two:

Feature	kube-lego	cert-manager
Configuration	Annotations on Ingress resources	CRDs
CAs	ACME	ACME, signing keypair
Kubernetes	v1.2 - v1.8	v1.7+
Debugging	Look at logs	Kubernetes Events API
Multi-tenancy	Not supported	Supported
Distinct issuance sources per Certificate	Not supported	Supported
Ingress controller support (ACME)	GCE, nginx	All

This guide will walk through how you can safely migrate your kube-lego installation to cert-manager, without service interruption.

By the end of the guide, we should have:

1. Scaled down and removed kube-lego
2. Installed cert-manager
3. Migrated ACME private key to cert-manager
4. Created an ACME ClusterIssuer using this private key, to issue certificates throughout your cluster
5. Configured cert-manager's *ingress-shim* to automatically provision Certificate resources for all Ingress resources with the `kubernetes.io/tls-acme: "true"` annotation, using the ClusterIssuer we have created
6. Verified that the cert-manager installation is working

2.2.1.1 1. Scale down kube-lego

Before we begin deploying cert-manager, it is best we scale our kube-lego deployment down to 0 replicas. This will prevent the two controllers potentially 'fighting' each other. If you deployed kube-lego using the official deployment YAMLs, a command like so should do:

```
$ kubectl scale deployment kube-lego \
    --namespace kube-lego \
    --replicas=0
```

You can then verify your kube-lego pod is no longer running with:

```
$ kubectl get pods --namespace kube-lego
```

2.2.1.2 2. Deploy cert-manager

cert-manager should be deployed using Helm, according to our official *Getting started* guide. No special steps are required here. We will return to this deployment at the end of this guide and perform an upgrade of some of the CLI flags we deploy cert-manager with however.

Please take extra care to ensure you have configured RBAC correctly when deploying Helm and cert-manager - there are some nuances described in our deploying document!

2.2.1.3 3. Obtaining your ACME account private key

In order to continue issuing and renewing certificates on your behalf, we need to migrate the user account private key that kube-lego has created for you over to cert-manager.

Your ACME user account identity is a private key, stored in a secret resource. By default, kube-lego will store this key in a secret named `kube-lego-account` in the same namespace as your kube-lego Deployment. You may have overridden this value when you deploy kube-lego, in which case the secret name to use will be the value of the `LEGO_SECRET_NAME` environment variable.

You should download a copy of this secret resource and save it in your local directory:

```
$ kubectl get secret kube-lego-account -o yaml \
    --namespace kube-lego \
    --export > kube-lego-account.yaml
```

Once saved, open up this file and change the `metadata.name` field to something more relevant to cert-manager. For the rest of this guide, we'll assume you chose `letsencrypt-private-key`.

Once done, we need to create this new resource in the `kube-system` namespace. By default, cert-manager stores supporting resources for ClusterIssuers in the namespace that it is running in, and we used `kube-system` when deploying cert-manager above. You should change this if you have deployed cert-manager into a different namespace.

```
$ kubectl create -f kube-lego-account.yaml \
    --namespace kube-system
```

2.2.1.4 4. Creating an ACME ClusterIssuer using your old ACME account

We need to create a ClusterIssuer which will hold information about the ACME account previously registered via kube-lego. In order to do so, we need two more pieces of information from our old kube-lego deployment: the server URL of the ACME server, and the email address used to register the account.

Both of these bits of information are stored within the kube-lego ConfigMap.

To retrieve them, you should be able to `get` the ConfigMap using `kubectl`:

```
$ kubectl get configmap kube-lego -o yaml \
    --namespace kube-lego \
    --export
```

Your email address should be shown under the `.data.lego.email` field, and the ACME server URL under `.data.lego.url`.

For the purposes of this guide, we will assume the lego email is `user@example.com` and the URL `https://acme-staging-v02.api.letsencrypt.org/directory`.

Now that we have migrated our private key to the new Secret resource, as well as obtaining our ACME email address and URL, we can create a ClusterIssuer resource!

Create a file named `cluster-issuer.yaml`:

```
1  apiVersion: certmanager.k8s.io/v1alpha1
2  kind: ClusterIssuer
3  metadata:
4    # Adjust the name here accordingly
5    name: letsencrypt-staging
6  spec:
7    acme:
8      # The ACME server URL
9      server: https://acme-staging-v02.api.letsencrypt.org/directory
10     # Email address used for ACME registration
11     email: user@example.com
12     # Name of a secret used to store the ACME account private key from step 3
```

(continues on next page)

(continued from previous page)

```
    privateKeySecretRef:
      name: letsencrypt-private-key
    # Enable the HTTP-01 challenge provider
    http01: {}
```

We then submit this file to our Kubernetes cluster:

```
$ kubectl create -f cluster-issuer.yaml
```

You should be able to verify the ACME account has been verified successfully:

```
$ kubectl describe clusterissuer letsencrypt-staging
Name:         letsencrypt-staging
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  certmanager.k8s.io/v1alpha1
Kind:         ClusterIssuer
Metadata:
  Cluster Name:
  Creation Timestamp:  2017-11-30T22:33:40Z
  Generation:          0
  Resource Version:    4450170
  Self Link:           /apis/certmanager.k8s.io/v1alpha1/letsencrypt-staging
  UID:                 83d04e6b-d61e-11e7-ac26-42010a840044
Spec:
  Acme:
    Email:  user@example.com
    Http 01:
    Private Key Secret Ref:
      Key:
      Name:  letsencrypt-private-key
    Server:  https://acme-staging-v02.api.letsencrypt.org/directory
Status:
  Acme:
    Uri:  https://acme-staging-v02.api.letsencrypt.org/acme/acct/11217539
  Conditions:
    Last Transition Time:  2018-04-12T17:32:30Z
    Message:               The ACME account was registered with the ACME server
    Reason:                ACMEAccountRegistered
    Status:                True
    Type:                  Ready
```

2.2.1.5 5. Configuring ingress-shim to use our new ClusterIssuer by default

Now that our ClusterIssuer is ready to issue certificates, we have one last thing to do: we must reconfigure ingress-shim (deployed as part of cert-manager) to automatically create Certificate resources for all Ingress resources it finds with appropriate annotations.

More information on the role of ingress-shim can be found *in the docs*, but for now we can just run a `helm upgrade` in order to add a few additional flags. Assuming you've named your ClusterIssuer `letsencrypt-staging` (as above), run:

```
helm upgrade cert-manager \
    stable/cert-manager \
```

(continues on next page)

(continued from previous page)

```
    --namespace kube-system \
    --set ingressShim.defaultIssuerName=letsencrypt-staging \
    --set ingressShim.defaultIssuerKind=ClusterIssuer
```

You should see the cert-manager pod be re-created, and once started it should automatically create Certificate resources for all of your ingresses that previously had kube-lego enabled.

2.2.1.6 6. Verify each ingress now has a corresponding Certificate

Before we finish, we should make sure there is now a Certificate resource for each ingress resource you previously enabled kube-lego on.

You should be able to check this by running:

```
$ kubectl get certificates --all-namespaces
```

There should be an entry for each ingress in your cluster with the kube-lego annotation.

We can also verify that cert-manager has 'adopted' the old TLS certificates by viewing the logs for cert-manager:

```
$ kubectl logs -n kube-system -l app=cert-manager -c cert-manager
...
I1025 21:54:02.869269       1 sync.go:206] Certificate my-example-certificate␣
↪scheduled for renewal in 292 hours
```

Here we can see cert-manager has verified the existing TLS certificate and scheduled it to be renewed in 292h time.

2.2.2 Securing nginx-ingress with Let's Encrypt

This guide talks you through securing a website exposed with nginx-ingress using a Certificate issued by Let's Encrypt.

2.2.2.1 Prerequisites

First, you should make sure you have properly configured and deployed nginx-ingress and at least one service is available **publicly** via the ingress controllers external IP address.

There's official deployment documentation in the official repository, or you can alternatively use Helm to deploy and manage your nginx-ingress installation.

2.2.3 Issuing an ACME certificate using DNS validation

Todo: This guide needs rewriting to be clearer, splitting into sections and potentially rewriting altogether.

cert-manager can be used to obtain certificates from a CA using the ACME protocol. The ACME protocol supports various challenge mechanisms which are used to prove ownership of a domain so that a valid certificate can be issued for that domain.

One such challenge mechanism is DNS-01. With a DNS-01 challenge, you prove ownership of a domain by proving you control its DNS records. This is done by creating a TXT record with specific content that proves you have control of the domains DNS records.

The following Issuer defines the necessary information to enable DNS validation. You can read more about the Issuer resource in the *Issuer reference docs*.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: letsencrypt-staging
  namespace: default
spec:
  acme:
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    email: user@example.com

    # Name of a secret used to store the ACME account private key
    privateKeySecretRef:
      name: letsencrypt-staging

    # ACME DNS-01 provider configurations
    dns01:

      # Here we define a list of DNS-01 providers that can solve DNS challenges
      providers:

        - name: prod-dns
          clouddns:
            # A secretKeyRef to a google cloud json service account
            serviceAccountSecretRef:
              name: clouddns-service-account
              key: service-account.json
            # The project in which to update the DNS zone
            project: gcloud-prod-project

        - name: cf-dns
          cloudflare:
            email: user@example.com
            # A secretKeyRef to a cloudflare api key
            apiKeySecretRef:
              name: cloudflare-api-key
              key: api-key.txt
```

We have specified the ACME server URL for Let's Encrypt's staging environment. The staging environment will not issue trusted certificates but is used to ensure that the verification process is working properly before moving to production. Let's Encrypt's production environment imposes much stricter rate limits, so to reduce the chance of you hitting those limits it is highly recommended to start by using the staging environment. To move to production, simply create a new Issuer with the URL set to `https://acme-v02.api.letsencrypt.org/directory`.

The first stage of the ACME protocol is for the client to register with the ACME server. This phase includes generating an asymmetric key pair which is then associated with the email address specified in the Issuer. Make sure to change this email address to a valid one that you own. It is commonly used to send expiry notices when your certificates are coming up for renewal. The generated private key is stored in a Secret named `letsencrypt-staging`.

The `dns01` stanza contains a list of DNS-01 providers that can be used to solve DNS challenges. Our Issuer defines two providers. This gives us a choice of which one to use when obtaining certificates.

More information about the DNS provider configuration, including a list of supported providers, can be found *in the dns01 reference docs*.

Once we have created the above Issuer we can use it to obtain a certificate.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  issuerRef:
    name: letsencrypt-staging
  commonName: '*.example.com'
  dnsNames:
  - example.com
  - foo.com
  acme:
    config:
    - dns01:
        provider: prod-dns
      domains:
      - '*.example.com'
      - example.com
    - dns01:
        provider: cf-dns
      domains:
      - foo.com
```

The Certificate resource describes our desired certificate and the possible methods that can be used to obtain it. You can obtain certificates for wildcard domains just like any other. Make sure to wrap wildcard domains with asterisks in your YAML resources, to avoid formatting issues. If you specify both `example.com` and `*.example.com` on the same Certificate, it will take slightly longer to perform validation as each domain will have to be validated one after the other. You can learn more about the Certificate resource in the *reference docs*. If the certificate is obtained successfully, the resulting key pair will be stored in a secret called `example-com-tls` in the same namespace as the Certificate.

The certificate will have a common name of `*.example.com` and the Subject Alternative Names (SANs) will be `*.example.com`, `example.com` and `foo.com`.

In our Certificate we have referenced the `letsencrypt-staging` Issuer above. The Issuer must be in the same namespace as the Certificate. If you want to reference a ClusterIssuer, which is a cluster-scoped version of an Issuer, you must add `kind: ClusterIssuer` to the `issuerRef` stanza.

For more information on ClusterIssuers, read the *ClusterIssuer reference docs*.

The `acme` stanza defines the configuration for our ACME challenges. Here we have defined the configuration for our DNS challenges which will be used to verify domain ownership. For each domain mentioned in a `dns01` stanza, cert-manager will use the provider's credentials from the referenced Issuer to create a TXT record called `_acme-challenge`. This record will then be verified by the ACME server in order to issue the certificate. Once domain ownership has been verified, any cert-manager affected records will be cleaned up.

Note: It is your responsibility to ensure the selected provider is authoritative for your domain.

After creating the above Certificate, we can check whether it has been obtained successfully using `kubectl describe`:

```
$ kubectl describe certificate example-com
Events:
  Type    Reason          Age      From          Message
```

(continues on next page)

(continued from previous page)

```
  ----    ------           ----     ----           -------
  Normal  CreateOrder      57m      cert-manager   Created new ACME order, attempting␣
↪validation...
  Normal  DomainVerified   55m      cert-manager   Domain "*.example.com" verified with
↪"dns-01" validation
  Normal  DomainVerified   55m      cert-manager   Domain "example.com" verified with
↪"dns-01" validation
  Normal  DomainVerified   55m      cert-manager   Domain "foo.com" verified with "dns-
↪01" validation
  Normal  IssueCert        55m      cert-manager   Issuing certificate...
  Normal  CertObtained     55m      cert-manager   Obtained certificate from ACME server
  Normal  CertIssued       55m      cert-manager   Certificate issued successfully
```

You can also check whether issuance was successful with `kubectl get secret example-com-tls -o yaml`. You should see a base64 encoded signed TLS key pair.

Once our certificate has been obtained, cert-manager will periodically check its validity and attempt to renew it if it gets close to expiry. cert-manager considers certificates to be close to expiry when the 'Not After' field on the certificate is less than the current time plus 30 days.

2.2.4 Issuing an ACME certificate using HTTP validation

Todo: This guide needs rewriting to be clearer, splitting into sections and potentially rewriting altogether.

cert-manager can be used to obtain certificates from a CA using the ACME protocol. The ACME protocol supports various challenge mechanisms which are used to prove ownership of a domain so that a valid certificate can be issued for that domain.

One such challenge mechanism is the HTTP-01 challenge. With a HTTP-01 challenge, you prove ownership of a domain by ensuring that a particular file is present at the domain. It is assumed that you control the domain if you are able to publish the given file under a given path.

The following Issuer defines the necessary information to enable HTTP validation. You can read more about the Issuer resource in the *Issuer reference docs*.

```
1  apiVersion: certmanager.k8s.io/v1alpha1
2  kind: Issuer
3  metadata:
4    name: letsencrypt-staging
5    namespace: default
6  spec:
7    acme:
8      # The ACME server URL
9      server: https://acme-staging-v02.api.letsencrypt.org/directory
10     # Email address used for ACME registration
11     email: user@example.com
12     # Name of a secret used to store the ACME account private key
13     privateKeySecretRef:
14       name: letsencrypt-staging
15     # Enable the HTTP-01 challenge provider
16     http01: {}
```

We have specified the ACME server URL for Let's Encrypt's staging environment. The staging environment will not issue trusted certificates but is used to ensure that the verification process is working properly before moving to

production. Let's Encrypt's production environment imposes much stricter rate limits, so to reduce the chance of you hitting those limits it is highly recommended to start by using the staging environment. To move to production, simply create a new Issuer with the URL set to `https://acme-v02.api.letsencrypt.org/directory`.

The first stage of the ACME protocol is for the client to register with the ACME server. This phase includes generating an asymmetric key pair which is then associated with the email address specified in the Issuer. Make sure to change this email address to a valid one that you own. It is commonly used to send expiry notices when your certificates are coming up for renewal. The generated private key is stored in a Secret named `letsencrypt-staging`.

The presence of the `http01` field simply enables the HTTP-01 challenge for this Issuer. No further configuration is necessary or currently possible.

Once we have created the above Issuer we can use it to obtain a certificate.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  issuerRef:
    name: letsencrypt-staging
  commonName: example.com
  dnsNames:
  - www.example.com
  acme:
    config:
    - http01:
        ingressClass: nginx
      domains:
      - example.com
    - http01:
        ingress: my-ingress
      domains:
      - www.example.com
```

The Certificate resource describes our desired certificate and the possible methods that can be used to obtain it. You can learn more about the Certificate resource in the *reference docs*. If the certificate is obtained successfully, the resulting key pair will be stored in a secret called `example-com-tls` in the same namespace as the Certificate.

The certificate will have a common name of `example.com` and the Subject Alternative Names (SANs) will be `example.com` and `www.example.com`.

In our Certificate we have referenced the `letsencrypt-staging` Issuer above. The Issuer must be in the same namespace as the Certificate. If you want to reference a ClusterIssuer, which is a cluster-scoped version of an Issuer, you must add `kind: ClusterIssuer` to the `issuerRef` stanza.

For more information on ClusterIssuers, read the *ClusterIssuer reference docs*.

The `acme` stanza defines the configuration for our ACME challenges. Here we have defined the configuration for our HTTP-01 challenges which will be used to verify domain ownership. To verify ownership of each domain mentioned in an `http01` stanza, cert-manager will create a Pod, Service and Ingress that exposes an HTTP endpoint that satisfies the HTTP-01 challenge.

The fields `ingress` and `ingressClass` in the `http01` stanza can be used to control how cert-manager interacts with Ingress resources:

- If the `ingress` field is specified, then an Ingress resource with the same name in the same namespace as the Certificate must already exist and it will be modified only to add the appropriate rules to solve the challenge.

This field is useful for the GCLB ingress controller, as well as a number of others, that assign a single public IP address for each ingress resource. Without manual intervention, creating a new ingress resource would cause any challenges to fail.

- If the `ingressClass` field is specified, a new ingress resource with a randomly generated name will be created in order to solve the challenge. This new resource will have an annotation with key `kubernetes.io/ingress.class` and value set to the value of the `ingressClass` field. This works for the likes of the NGINX ingress controller.
- If neither are specified, new ingress resources will be created with a randomly generated name, but they will not have the ingress class annotation set.
- If both are specified, then the `ingress` field will take precedence.

Once domain ownership has been verified, any cert-manager affected resources will be cleaned up or deleted.

Note: It is your responsibilty to point each domain name at the correct IP address for your ingress controller.

After creating the above Certificate, we can check whether it has been obtained successfully using `kubectl describe`:

```
$ kubectl describe certificate example-com
Events:
  Type    Reason          Age      From          Message
  ----    ------          ----     ----          -------
  Normal  CreateOrder     57m      cert-manager  Created new ACME order, attempting␣
↪validation...
  Normal  DomainVerified  55m      cert-manager  Domain "example.com" verified with
↪"http-01" validation
  Normal  DomainVerified  55m      cert-manager  Domain "www.example.com" verified␣
↪with "http-01" validation
  Normal  IssueCert       55m      cert-manager  Issuing certificate...
  Normal  CertObtained    55m      cert-manager  Obtained certificate from ACME server
  Normal  CertIssued      55m      cert-manager  Certificate issued successfully
```

You can also check whether issuance was successful with `kubectl get secret example-com-tls -o yaml`. You should see a base64 encoded signed TLS key pair.

Once our certificate has been obtained, cert-manager will periodically check its validity and attempt to renew it if it gets close to expiry. cert-manager considers certificates to be close to expiry when the 'Not After' field on the certificate is less than the current time plus 30 days.

2.3 CA Issuer Tutorials

This section contains tutorials that specifically utilise the CA Issuer. They are designed to help teach the underlying concepts of cert-manager whilst also helping 'quickstart' common use-cases for the project.

2.3.1 Creating a simple CA based issuer

cert-manager can be used to obtain certificates using an arbitrary signing key pair stored in a Kubernetes Secret resource.

This guide will show you how to configure and create a CA based issuer, backed by a signing key pair stored in a Secret resource.

2.3.1.1 1. (Optional) Generate a signing key pair

The CA Issuer does not automatically create and manage a signing key pair for you. As a result, you will need to either supply your own or generate a self signed CA using a tool such as openssl or cfssl.

This guide will explain how to generate a new signing key pair, however you can substitute it for your own so long as it has the `CA` flag set.

```
# Generate a CA private key
$ openssl genrsa -out ca.key 2048

# Create a self signed Certificate, valid for 10yrs with the 'signing' option set
$ openssl req -x509 -new -nodes -key ca.key -subj "/CN=${COMMON_NAME}" -days 3650 -
↪reqexts v3_req -extensions v3_ca -out ca.crt
```

The output of these commands will be two files, `ca.key` and `ca.crt`, the key and certificate for your signing key pair. If you already have your own key pair, you should name the private key and certificate `ca.key` and `ca.crt` respectively.

2.3.1.2 2. Save the signing key pair as a Secret

We are going to create an Issuer that will use this key pair to generate signed certificates. You can read more about the Issuer resource in *the Issuer reference docs*. To allow the Issuer to reference our key pair we will store it in a Kubernetes Secret resource.

Issuers are namespaced resources and so they can only reference Secrets in their own namespace. We will therefore put the key pair into the same namespace as the Issuer. We could alternatively create a *ClusterIssuer*, a cluster-scoped version of an Issuer. For more information on ClusterIssuers, read the *ClusterIssuer reference documentation*.

The following command will create a Secret containing a signing key pair in the default namespace:

```
kubectl create secret tls ca-key-pair \
   --cert=ca.crt \
   --key=ca.key \
   --namespace=default
```

2.3.1.3 3. Creating an Issuer referencing the Secret

We can now create an Issuer referencing the Secret resource we just created:

```
1 apiVersion: certmanager.k8s.io/v1alpha1
2 kind: Issuer
3 metadata:
4   name: ca-issuer
5   namespace: default
6 spec:
7   ca:
8     secretName: ca-key-pair
```

We are now ready to obtain certificates!

2.3.1.4 4. Obtain a signed Certificate

We can now create the following Certificate resource which specifies the desired certificate. You can read more about the Certificate resource in *the reference docs*.

```yaml
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  issuerRef:
    name: ca-issuer
    # We can reference ClusterIssuers by changing the kind here.
    # The default value is Issuer (i.e. a locally namespaced Issuer)
    kind: Issuer
  commonName: example.com
  organization:
  - Example CA
  dnsNames:
  - example.com
  - www.example.com
```

In order to use the Issuer to obtain a Certificate, we must create a Certificate resource in the **same namespace as the Issuer**, as an Issuer is a namespaced resource. We could alternatively create a *ClusterIssuer* if we wanted to reuse the signing key pair across multiple namespaces.

Once we have created the Certificate resource, cert-manager will attempt to use the Issuer `ca-issuer` to obtain a certificate. If successful, the certificate will be stored in a Secret resource named `example-com-tls` in the same namespace as the Certificate resource (`default`).

Note that since we have specified the `commonName` field, `example.com` will be the common name for our certificate and both the common name and all the elements of the `dnsNames` array will be Subject Alternative Names (SANs). If we had not specified the common name then the first element of the `dnsNames` list would be used as the common name and all elements of the `dnsNames` list would also be SANs.

After creating the above Certificate, we can check whether it has been obtained successfully like so:

```
$ kubectl describe certificate example-com
Events:
  Type     Reason                 Age              From                     Message
  ----     ------                 ----             ----                     -------
  Warning  ErrorCheckCertificate  26s              cert-manager-controller  Error␣
↪checking existing TLS certificate: secret "example-com-tls" not found
  Normal   PrepareCertificate     26s              cert-manager-controller  Preparing␣
↪certificate with issuer
  Normal   IssueCertificate       26s              cert-manager-controller  Issuing␣
↪certificate...
  Normal   CertificateIssued      25s              cert-manager-controller ␣
↪Certificate issued successfully
```

You can also check whether issuance was successful with `kubectl get secret example-com-tls -o yaml`. You should see a base64 encoded signed TLS key pair.

Once the certificate has been obtained, cert-manager will keep checking its validity and attempt to renew it if it gets close to expiry. cert-manager considers certificates to be close to expiry when the 'Not After' field on the certificate is less than the current time plus 30 days. For CA based Issuers, cert-manager will issue certificates with the 'Not After' field set to the current time plus 365 days.

2.4 Vault Issuer Tutorials

This section contains tutorials that specifically utilise the Vault Issuer. They are designed to help teach the underlying concepts of cert-manager whilst also helping 'quickstart' common use-cases for the project.

2.4.1 Vault Installation

2.4.1.1 Installing Vault

Vault installation is a complex subject. For a thorough tour of the subject you can read the official HashiCorp Vault documentation.

2.4.1.2 Vault PKI Backend

The PKI Secrets Engine needs to be initialized for cert-manager to be able to generate certificate. The official Vault documentation can be found here.

2.4.2 Vault Authentication with a AppRole

This Vault authentication method uses a Vault AppRole.

The secret ID of the AppRole is stored in a secret.

Here an example of a secret containing the secretId of the AppRole:

```
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: cert-manager-vault-approle
  namespace: default
data:
  secretId: "MDI..."
```

Where the secretId is the base 64 encoded value of the appRole *secretId* giving access to the pki backend in Vault.

We can now create a cluster issuer referencing this secret:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: vault-issuer
  namespace: default
spec:
  vault:
    path: pki_int/sign/example-dot-com
    server: https://vault
    caBundle: <base64 encoded caBundle PEM file>
    auth:
      appRole:
        path: approle
        roleId: "291b9d21-8ff5-..."
        secretRef:
```

(continues on next page)

(continued from previous page)

```
          name: cert-manager-vault-approle
          key: secretId
```

Where *path* is the Vault role path of the PKI backend and *server* is the Vault server base URL. The *path* MUST USE the vault `sign` endpoint. The Vault appRole credentials are supplied as the Vault authentication method using the appRole created in Vault. The secretRef references the Kubernetes secret created previously. More specifically, the field *name* is the Kubernetes secret name and *key* is the name given as the key value that store the *secretId*. The optional attribute *path* specifies where the AppRole authentication is mounted in Vault. The attribute *path* default value is *approle*.

An optional base64 encoded *caBundle* in PEM format can be provided to validate the TLS connection to the Vault Server. When *caBundle* is set it replaces the CA bundle inside the container running cert-manager. This parameter as no effect if the connection used is in plain HTTP.

Once we have created the above Issuer we can use it to obtain a certificate.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  issuerRef:
    name: vault-issuer
  commonName: example.com
  dnsNames:
  - www.example.com
```

The Certificate resource describes our desired certificate and the possible methods that can be used to obtain it. You can learn more about the Certificate resource in the *reference docs*. If the certificate is obtained successfully, the resulting key pair will be stored in a secret called `example-com-tls` in the same namespace as the Certificate.

The certificate will have a common name of `example.com` and the Subject Alternative Names (SANs) will be `example.com` and `www.example.com`.

In our Certificate we have referenced the `vault-issuer` Issuer above. The Issuer must be in the same namespace as the Certificate. If you want to reference a ClusterIssuer, which is a cluster-scoped version of an Issuer, you must add `kind: ClusterIssuer` to the `issuerRef` stanza.

For more information on ClusterIssuers, read the *ClusterIssuer reference docs*.

2.4.3 Vault Authentication with a Token

This Vault authentication method uses a plain token. A Vault token is generated by one of the many authentication backend supported by Vault. Tokens in Vault have expiration and need to be refreshed. You need to be aware that cert-manager do not refresh these tokens. Another process must be put in place to keep them from expiring.

For testing purpose a root token which do not expire is generated at Vault installation time. **WARNING: a root token should only be used for testing purpose only**.

Please refer to the official token documentation for all the details.

Here an example of a secret Kubernetes resource containing the Vault token:

```
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: cert-manager-vault-token
  namespace: kube-system
data:
  token: "MjI..."
```

Where the token value is the base 64 encoded value of the token giving access to the PKI backend in Vault.

We can now create an issuer referencing this secret:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: vault-issuer
  namespace: default
spec:
  vault:
    auth:
      tokenSecretRef:
        name: cert-manager-vault-token
        key: token
    path: pki_int/sign/example-dot-com
    server: https://vault
    caBundle: <base64 encoded caBundle PEM file>
```

Where *path* is the Vault role path of the PKI backend and *server* is the Vault server base URL. The secret created previously is referenced in the issuer with its *name* and *key* corresponding to the name of the Kubernetes secret and the property name containing the token value respectively.

An optional base64 encoded *caBundle* in PEM format can be provided to validate the TLS connection to the Vault Server. When *caBundle* is set it replaces the CA bundle inside the container running cert-manager. This parameter as no effect if the connection used is in plain HTTP.

Once we have created the above Issuer we can use it to obtain a certificate.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  issuerRef:
    name: vault-issuer
  commonName: example.com
  dnsNames:
  - www.example.com
```

The Certificate resource describes our desired certificate and the possible methods that can be used to obtain it. You can learn more about the Certificate resource in the *reference docs*. If the certificate is obtained successfully, the resulting key pair will be stored in a secret called `example-com-tls` in the same namespace as the Certificate.

The certificate will have a common name of `example.com` and the Subject Alternative Names (SANs) will be `example.com` and `www.example.com`.

In our Certificate we have referenced the `vault-issuer` Issuer above. The Issuer must be in the same namespace

as the Certificate. If you want to reference a ClusterIssuer, which is a cluster-scoped version of an Issuer, you must add `kind: ClusterIssuer` to the `issuerRef` stanza.

For more information on ClusterIssuers, read the *ClusterIssuer reference docs*.

2.5 Venafi Issuer Tutorials

This section contains tutorials that utilise the Venafi Issuer. They are designed to help teach the underlying concepts of cert-manager whils also helping 'quickstart' common use-cases for the project.

2.5.1 Setting up a Venafi Cloud or TPP Issuer

The Venafi issuer is an extension which supports certificate management from Venafi Cloud and Venafi Trust Protection Platform.

2.5.1.1 Deploying cert-manager

Note: Please also see the 'Getting Started' guide on the left for more info on how to deploy cert-manager.

Note: This guide assumes you already have a functioning Kubernetes cluster of version 1.9 or greater.

You can run the following to deploy cert-manager with the Venafi integration for the first time:

```
kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/venafi/
↪contrib/manifests/cert-manager/with-rbac.yaml
```

Note: This step only needs to be performed once!

Note: Please verify that no errors were output when you run this command.

2.5.1.2 Creating Venafi Cloud issuer

Register your account at https://api.venafi.cloud/login and get API key there.

Create a secret containing your authentication credentials for the issuer to use (in this example, the Issuer will utilise Venafi Cloud and will only issue certificates in the `default` namespace).

```
kubectl create secret generic cloudsecret --from-literal=apikey='YOUR_CLOUD_API_KEY_
↪HERE'
```

Create the issuer, referencing the secret we just created:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: cloud-venafi-issuer
spec:
  venafi:
    zone: "DevOps"
    cloud:
      apiKeySecretRef:
        name: cloudsecret
        key: apikey
```

You can create multiple issuers pointing to different Venafi Cloud zones, or even have 1 issuer pointing to Venafi Platform and another pointing to Venafi Cloud.

We can then create a certificate resource that utilises this newly configured issuer:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: cert4-venafi-localhost
spec:
  commonName: cert4.venafi.localhost
  secretName: cert4-venafi-localhost
  issuerRef:
    name: cloud-venafi-issuer
```

To see the full list of options available on the Certificate resource, take a look at the `API reference documentation`.

2.5.1.3 Creating Venafi Platform issuer

Similar to how we created credentials and an Issuer resource for TPP Cloud above, we can also create Issuers for Venafi TPP instances.

Again, you can have multiple Issuer's for different Venafi zones, and even run Venafi Cloud Issuers alongside Venafi TPP Issuers.

Requirements for Venafi Platform policy

1. You **must** allow "User Provided CSRs" as part of your TPP policy, as this is the only type supported by the underlying `vcert` library we use.
2. MSCA configuration should have http URI set before the ldap URI in X509 extensions, otherwise NGINX ingress controller couldn't get certificate chain from URL and OSCP will not work. Example: TODO: verify this/make it clearer

```
X509v3 extensions:
    X509v3 Subject Alternative Name:
    DNS:test-cert-manager1.venqa.venafi.com}}
    X509v3 Subject Key Identifier: }}
    61:5B:4D:40:F2:CF:87:D5:75:5E:58:55:EF:E8:9E:02:9D:E1:81:8E}}
    X509v3 Authority Key Identifier: }}
    keyid:3C:AC:9C:A6:0D:A1:30:D4:56:A7:3D:78:BC:23:1B:EC:B4:7B:4D:75}}X509v3 CRL␣
↪Distribution Points:Full Name:
    URI:http://qavenafica.venqa.venafi.com/CertEnroll/QA%20Venafi%20CA.crl}}
    URI:ldap:///CN=QA%20Venafi%20CA,CN=qavenafica,CN=CDP,CN=Public%20Key%20Services,
↪CN=Services,CN=Configuration,DC=venqa,DC=venafi,DC=com?certificateRevocationList?
↪base?objectClass=cRLDistributionPoint}}{{Authority Information Access: }}
```

(continues on next page)

(continued from previous page)

```
    CA Issuers - URI:http://qavenafica.venqa.venafi.com/CertEnroll/qavenafica.venqa.
↪venafi.com_QA%20Venafi%20CA.crt}}
    CA Issuers - URI:ldap:///CN=QA%20Venafi%20CA,CN=AIA,CN=Public%20Key%20Services,
↪CN=Services,CN=Configuration,DC=venqa,DC=venafi,DC=com?cACertificate?base?
↪objectClass=certificationAuthority}}
```

3. Option in Venafi Platform CA configuration template "Automatically include CN as DNS SAN" should be set to true. (TODO this shouldn't be a requirement)

Create a secret with Venafi Platform credentials:

Like before, we create a Secret resource containing our Venafi TPP credentials:

```
kubectl create secret generic tppsecret \
    --from-literal=user=admin \
    --from-literal=password=tpppassword
```

Create Venafi Platform issuer

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: tpp-venafi-issuer
spec:
  zone: devops\cert-manager # must exist in the TPP console
  venafi:
    tpp:
      url: https://tpp.venafi.example/vedsdk
      credentialsRef:
        name: tppsecret
```

Create a certificate

Just the same as before, we can create a Certificate resource that utilises the TPP Issuer we just created:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: hellodemo-venafi-localhost
spec:
  commonName: hellodemo.venafi.localhost
  secretName: hellodemo-venafi-localhost
  issuerRef:
    name: tppvenafiissuer
```

CHAPTER 3

Administrative tasks

This section contains documentation about common administrative tasks for cert-manager.

Explore the sections listed below to find more information on how to run and manage a cert-manager deployment as an administrator.

3.1 Resource Validation Webhook

In order to provide advanced resource validation, cert-manager includes a ValidatingWebhookConfiguration which is deployed into the cluster as its own pod.

This feature requires Kubernetes 1.9 or greater. If you disable the webhook component, cert-manager will still perform the same resource validation however will not reject 'create' events when submitting resources to the API server.

The webhook component is disabled by default, and must be enabled when installing with the helm chart, or installed as an additional component if using the static manifests.

3.1.1 Enabling the webhook component

3.1.1.1 With Helm

To enable the component when using Helm, you must first ensure the namespace that you deploy cert-manager into has the label `certmanager.k8s.io/disable-validation: "true"`.

You can add this label like so:

```
$ kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true
```

Note: New installations of cert-manager with Helm v2.10 and later will not require this additional step

You can then proceed to upgrade your Helm deployment as usual, adding one additional flag:

```
$ helm upgrade cert-manager stable/cert-manager \
      --reuse-values \
      --set webhook.enabled=true
```

3.1.1.2 With static manifests

When installing using the static manifests, the webhook component is installed as a separate set of manifests.

You can find the manifests for the webhook in the deploy directory.

3.1.2 FAQ

3.1.2.1 TLS Configuration

The ValidatingWebhookConfiguration resource requires that the webhook server uses TLS.

cert-manager uses a commbination of the SelfSigned and CA Issuer types to provision the resources required to do this.

In order to do this, when installing with the Helm chart or static deployment manifests, resource validation is **disabled** on the nammespace cert-manager is deployed into.

Note: If you have manually created the namespace that cert-manager is deployed into, you must ensure your namespace has the `certmanager.k8s.io/disable-validation: "true"` Label set on the Namespace resource. This is handled automatically when performing a `helm install` for the first time by use of an additional selector in the ValidatingWebhookConfiguration

---

1) First, a self-signed Issuer is created in order to issue self-signed certificates. You can see this named `cm-cert-manager-selfsign` in the output below.

2) Then, a Certificate resource referencing the self-signed Issuer is created. This certificate has `spec.isCA: true` set. It will be used as our root CA. You can see this named `cm-cert-manager-webhook-ca` in the output below.

3) Then another Issuer resource is created, this time a **CA** Issuer. This Issuer will issue certificates signed by the self-signed root CA created in (2). You can see this named `cm-cert-manager-webhook-ca` in the output below.

4) Finally, a second Certificate resource is created. This one will be used by the webhook to secure communication between the apiserver and the webhook! You can see this named `cm-cert-manager-webhook-tls` in the output below.

You can see the status of the certificates and issuers used for the webhook in your own cluster by running:

```
$ kubectl describe certificate --namespace cert-manager
Name:         cm-cert-manager-webhook-ca
Namespace:    cert-manager
Labels:       <none>
Annotations:  <none>
API Version:  certmanager.k8s.io/v1alpha1
Kind:         Certificate
Metadata:
  Cluster Name:
  Creation Timestamp:  2018-08-07T23:18:53Z
  Generation:          0
  Resource Version:    722
```

(continues on next page)

(continued from previous page)

```
  Self Link:           /apis/certmanager.k8s.io/v1alpha1/namespaces/cert-manager/
↪certificates/cm-cert-manager-webhook-ca
  UID:                 402722a2-9a98-11e8-bf3f-525400856e41
Spec:
  Common Name:  ca.webhook.cert-manager
  Is CA:        true
  Issuer Ref:
    Name:       cm-cert-manager-selfsign
  Secret Name:  cm-cert-manager-webhook-ca
Status:
  Conditions:
    Last Transition Time:  2018-08-07T23:18:57Z
    Message:               Certificate issued successfully
    Reason:                CertIssued
    Status:                True
    Type:                  Ready
Events:
  Type    Reason      Age   From          Message
  ----    ------      ----  ----          -------
  Normal  IssueCert   9m    cert-manager  Issuing certificate...
  Normal  CertIssued  9m    cert-manager  Certificate issued successfully


Name:         cm-cert-manager-webhook-tls
Namespace:    cert-manager
Labels:       <none>
Annotations:  <none>
API Version:  certmanager.k8s.io/v1alpha1
Kind:         Certificate
Metadata:
  Cluster Name:
  Creation Timestamp:  2018-08-07T23:18:53Z
  Generation:          0
  Resource Version:    738
  Self Link:           /apis/certmanager.k8s.io/v1alpha1/namespaces/cert-manager/
↪certificates/cm-cert-manager-webhook-tls
  UID:                 4021e81e-9a98-11e8-bf3f-525400856e41
Spec:
  Dns Names:
    cm-cert-manager-webhook
    cm-cert-manager-webhook.cert-manager
    cm-cert-manager-webhook.cert-manager.svc
  Is CA:  false
  Issuer Ref:
    Name:       cm-cert-manager-webhook
  Secret Name:  cm-cert-manager-webhook-tls
Status:
  Conditions:
    Last Transition Time:  2018-08-07T23:19:01Z
    Message:               Certificate issued successfully
    Reason:                CertIssued
    Status:                True
    Type:                  Ready
Events:
  Type     Reason          Age   From          Message
  ----     ------          ----  ----          -------
  Warning  IssuerNotReady  9m    cert-manager  Issuer cm-cert-manager-webhook not␣
↪ready
```

(continues on next page)

(continued from previous page)

```
  Normal   IssueCert       9m    cert-manager  Issuing certificate...
  Normal   CertIssued      9m    cert-manager  Certificate issued successfully


$ kubectl describe issuer --namespace cert-manager
Name:         cm-cert-manager-selfsign
Namespace:    cert-manager
Labels:       <none>
Annotations:  <none>
API Version:  certmanager.k8s.io/v1alpha1
Kind:         Issuer
Metadata:
  Cluster Name:
  Creation Timestamp:  2018-08-07T23:18:53Z
  Generation:          0
  Resource Version:    696
  Self Link:           /apis/certmanager.k8s.io/v1alpha1/namespaces/cert-manager/
↪issuers/cm-cert-manager-selfsign
  UID:                 402a07c1-9a98-11e8-bf3f-525400856e41
Spec:
  Self Signed:
Status:
  Conditions:
    Last Transition Time:  2018-08-07T23:18:55Z
    Message:
    Reason:                IsReady
    Status:                True
    Type:                  Ready
Events:                    <none>


Name:         cm-cert-manager-webhook-ca
Namespace:    cert-manager
Labels:       <none>
Annotations:  <none>
API Version:  certmanager.k8s.io/v1alpha1
Kind:         Issuer
Metadata:
  Cluster Name:
  Creation Timestamp:  2018-08-07T23:18:53Z
  Generation:          0
  Resource Version:    726
  Self Link:           /apis/certmanager.k8s.io/v1alpha1/namespaces/cert-manager/
↪issuers/cm-cert-manager-webhook-ca
  UID:                 402ea69e-9a98-11e8-bf3f-525400856e41
Spec:
  Ca:
    Secret Name:  cm-cert-manager-webhook-ca
Status:
  Conditions:
    Last Transition Time:  2018-08-07T23:18:58Z
    Message:               Signing CA verified
    Reason:                KeyPairVerified
    Status:                True
    Type:                  Ready
Events:
  Type     Reason           Age              From          Message
```

(continues on next page)

(continued from previous page)

```
  ----     ------            ----              ----           -------
  Warning  ErrGetKeyPair     9m                cert-manager   Error getting keypair for␣
↪CA issuer: secret "cm-cert-manager-webhook-ca" not found
  Warning  ErrInitIssuer     9m                cert-manager   Error initializing issuer:␣
↪secret "cm-cert-manager-webhook-ca" not found
  Warning  ErrGetKeyPair     9m (x6 over 9m)   cert-manager   Error getting keypair for␣
↪CA issuer: secret "cm-cert-manager-webhook-ca" not found
  Warning  ErrInitIssuer     9m (x6 over 9m)   cert-manager   Error initializing issuer:␣
↪secret "cm-cert-manager-webhook-ca" not found
  Normal   KeyPairVerified   9m (x2 over 9m)   cert-manager   Signing CA verified
```

3.1.2.2 Keeping Kubernetes PKI resources up to date

Once the root CA certificate has been provisioned, cert-manager also needs to update the Kubernetes API Server to give it a copy of the root CA in order to verify connections to the webhook component.

To do this, the `spec.caBundle` field on the `APIService` resource named `v1beta1.admission.certmanager.k8s.io` must be set to the root CA generated above, and the ValidatingWebhookConfiguration named `cert-manager-webhook` must have its own `caBundle` fields set to that of your Kubernetes API Server.

The cert-manager deployment manifests do this automatically by installing a Kubernetes CronJob resource. This CronJob will run every 24 hours and ensures that these resources are up to date.

The code for this component can be found at munnerz/apiextensions-ca-helper

3.2 Upgrading cert-manager

This section contains information on upgrading cert-manager. It also contains documents detailing breaking changes between cert-manager versions, and information on things to look out for when upgrading.

3.2.1 Upgrading with Helm

If you installed cert-manager using Helm, you can easily upgrade using the Helm CLI.

Note: Before upgrading, please read the relevant instructions at the links below for your from and to version.

Once you have read the relevant notes and taken any appropriate actions, you can begin the upgrade process like so - replacing `<release_name>` with the name of your Helm release for cert-manager (usually this is `cert-manager`):

```
$ helm repo update
$ helm upgrade <release_name> stable/cert-manager
```

This will upgrade you to the latest version of cert-manager, as listed in the official Helm charts repository.

Note: You can find out your release name using `helm list | grep cert-manager`.

3.2.2 Upgrading using static manifests

If you installed cert-manager using the static deployment manifests, you can upgrade them in a similar way to how you first installed them.

Note: Before upgrading, please read the relevant instructions at the links below for your from and to version.

Once you have read the relevant notes and taken any appropriate actions, you can begin the upgrade process like so - replacing `<version_number>` and `<namespace>` with the version number you want to install, and the namespace that cert-manager is deployed into respectively:

```
$ kubectl apply --namespace <namespace> -f \
    https://github.com/jetstack/cert-manager/blob/<version_number>/contrib/manifests/
↪cert-manager/with-rbac.yaml
```

Note: Please be sure to choose the correct variant of the static manifests, i.e. one of the `with-rbac` or `without-rbac` files, depending on your cluster configuration.

3.2.2.1 Upgrading from v0.2 to v0.3

During the v0.3 release, a number of breaking changes were made that require you to update either deployment configuration and runtime configuration (e.g. Certificate, Issuer and ClusterIssuer resources).

After reading these instructions, you should then proceed to upgrade cert-manager according to your deployment configuration (e.g. using `helm upgrade` if installing via Helm chart, or `kubectl apply` if installing with raw manifests).

A brief summary:

- Supporting resources for ClusterIssuers (e.g. signing CA certificates, or ACME account private keys) will now be stored in the same namespace as cert-manager, instead of kube-system in previous versions (#329, @munnerz)
- Switch to ConfigMaps instead of Endpoints for leader election (#327, @mikebryant)
- Removing support for ACMEv1 in favour of ACMEv2 (#309, @munnerz)
- Removing ingress-shim and compiling it into cert-manager itself (#502, @munnerz)
- Change to the default behaviour of ingress-shim. It now generates Certificates with the `ingressClass` field set instead of the `ingress` field. This will mean users of ingress controllers that assign a single IP to a single Ingress (e.g. the GCE ingress controller) will no longer work without adding a new annotation to your ingress resource.

3.2.2.1.1 Supporting resources for ClusterIssuers moving into the cert-manager namespace

In the past, the cert-manager controller was hard coded to look for supplemental resources, such as Secrets containing DNS provider credentials, in the kube-system namespace.

We now store these resources in the same namespace as the cert-manager pod itself runs within.

When upgrading, you should make sure to move any of these supplemental resources into the cert-manager deployment namespace, or otherwise deploy cert-manager into kube-system itself.

You can also change the 'cluster resource namespace' when deploying cert-manager:

With the helm chart: `--set clusterResourceNamespace=kube-system`.

Or if using the static deployment manifests, by adding the `--cluster-resource-namespace` flag to the `args` field of the cert-manager container.

3.2.2.1.2 Switch to ConfigMaps instead of Endpoints for leader election

cert-manager-controller performs leader election to allow you to run 'hot standby' replicas of cert-manager.

In the past, we used Endpoint resources to perform this election. The new best practice is to use ConfigMap resources in order to reduce API overhead in large clusters.

As such, v0.3 switches us to use ConfigMap resources for leader election.

During the upgrade, you should first scale your cert-manager-controller deployment to 0 to ensure no other replicas of cert-manager are running when the new v0.3 deployment starts:

```
kubectl scale --namespace <deployment-namespace> --replicas=0 deployment <cert-
↪manager-deployment-name>
```

3.2.2.1.3 Removing support for ACMEv1 in favour of ACMEv2

The ACME v2 specification is now in production with Let's Encrypt. In order to support this new spec, which includes support for wildcard certificates, we have removed support for the v1 protocol altogether.

If you have any ACME Issuer or ClusterIssuer resources, you should update the server fields of these to the new ACMEv2 endpoints.

For example, if you have a Let's Encrypt production issuer, you should update the server URL:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
...
spec:
  acme:
    # server: https://acme-v01.api.letsencrypt.org/directory
    server: https://acme-v02.api.letsencrypt.org/directory # we switch 'v01' to 'v02'
```

3.2.2.1.4 Removing ingress-shim and compiling it into cert-manager itself

In v0.3 we removed the ingress-shim component and instead now compile in its functionality into the main cert-manager binary.

This change also introduces a change to the way you configure default Issuers and ClusterIssuers at deployment time.

The deployment documentation has been updated accordingly, but instead of setting `ingressShim.extraArgs={--default-issuer-name=letsencrypt-pod}` there are now dedicated Helm chart fields:

```
--set ingressShim.defaultIssuerName=letsencrypt-prod \
--set ingressShim.defaultIssuerKind=ClusterIssuer
```

3.2.2.1.5 Change to the default behaviour of ingress-shim

In the past, when using ingress-shim, we set the `ingress` field on the Certificate resource to trigger cert-manager to edit the specified Ingress resource to solve the challenge.

The alternate option is to set the `ingressClass` field, which causes cert-manager to create temporary Ingress resources to solve the challenge. This behaviour provides better compatibility with ingress controllers like nginx-ingress.

In v0.3 we have changed the default behaviour of ingress-shim to set the `ingressClass` field instead of `ingress`.

This will cause validations for ingress controllers like ingress-gce to fail without additional configuration in your Ingress resources annotations.

Add the follow annotation to your Ingress resources if you are using the GCE ingress controller, in addition to the usual ingress-shim annotation(s):

```
certmanager.k8s.io/acme-http01-edit-in-place: "true"
```

3.2.2.2 Upgrading from v0.3 to v0.4

There are no special notes or considerations when upgrading from v0.3 to v0.4.

3.2.2.3 Upgrading from v0.4 to v0.5

Version 0.5 of cert-manager introduces a new 'webhook' component, which is used by the Kubernetes apiserver to validate our CRD resource types.

This should help in future to reduce errors caused by misconfigured Certificate and Issuer resources.

When upgrading from a previous release using Helm, it is **essential** that you perform one extra step before upgrading.

3.2.2.3.1 Disabling resource validation on the cert-manager namespace

Before upgrading, you should add the `certmanager.k8s.io/disable-validation: "true"` label to the `cert-manager` namespace.

This will allow the system resources that cert-manager requires to bootstrap TLS to be created in its own namespace.

CHAPTER 4

Reference documentation

This section contains detailed reference documentation about cert-manager's types and how it operates. It also includes some simple example configurations in order to help users activate advanced functionality of cert-manager.

Step by step user guides and tutorials can be found in the *tutorials* section.

4.1 Certificates

cert-manager has the concept of 'Certificates' that define a desired X.509 certificate. A Certificate is a namespaced resource that references an Issuer or ClusterIssuer for information on how to obtain the certificate.

A simple Certificate could be defined as:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: acme-crt
spec:
  secretName: acme-crt-secret
  dnsNames:
  - foo.example.com
  - bar.example.com
  acme:
    config:
    - ingressClass: nginx
      domains:
      - foo.example.com
      - bar.example.com
  issuerRef:
    name: letsencrypt-prod
    # We can reference ClusterIssuers by changing the kind here.
    # The default value is Issuer (i.e. a locally namespaced Issuer)
    kind: Issuer
```

This Certificate will tell cert-manager to attempt to use the Issuer named `letsencrypt-prod` to obtain a certificate key pair for the `foo.example.com` and `bar.example.com` domains. If successful, the resulting key and certificate will be stored in a secret named `acme-crt-secret` with keys of `tls.key` and `tls.crt` respectively. This secret will live in the same namespace as the `Certificate` resource.

The `dnsNames` field specifies a list of Subject Alternative Names to be associated with the certificate. If the `commonName` field is omitted, the first element in the list will be the common name.

The referenced Issuer must exist in the same namespace as the Certificate. A Certificate can alternatively reference a ClusterIssuer which is non-namespaced.

4.1.1 ACME Specific Certificate Config

When creating Certificate resources that reference an ACME Issuer, there are a number of extra configuration parameters that can be specified to influence how the Certificate will be obtained (notably, configuring the ACME challenge type).

Todo: Extend this document with information on configuring the ACME stanza on Certificate resources.

Todo: Make this document more prominent

4.1.2 Certificate Duration and Renewal Window

cert-manager Certificate resources also support custom validity durations and renewal windows.

Important: The backend service implementation can choose to generate a certificate with a different validity period than what is requested in the issuer.

Although the duration and renewal periods are specified on the Certificate resources, the corresponding Issuer or ClusterIssuer must support this.

The table below shows the support state of the different backend services used by issuer types:

Issuer	Description
ACME	Only 'renewBefore' supported
CA	Fully supported
Vault	Fully supported (although the requested duration must be lower than the configured Vault role's TTL)
Self Signed	Fully supported

The default duration for all certificates is 90 days and the default renewal windows is 30 days. This means that certificates are considered valid for 3 months and renewal will be attempted within 1 month of expiration.

The *duration* and *renewBefore* parameters must be given in the golang parseDuration string format.

4.1.2.1 Example Usage

Here an example of an issuer specifying the duration and renewal window.

The certificate from the previous section is extended with a validity period of 24 hours and to begin trying to renew 12 hours before the certificate expiration.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example
spec:
  secretName: example-tls
  duration: 24h
  renewBefore: 12h
  dnsNames:
  - foo.example.com
  - bar.example.com
  issuerRef:
    name: my-internal-ca
    kind: Issuer
```

4.2 Issuers

Issuers (and *ClusterIssuers*) represent a certificate authority from which signed x509 certificates can be obtained, such as Let's Encrypt. You will need at least one Issuer or ClusterIssuer in order to begin issuing certificates within your cluster.

An example of an Issuer type is ACME. A simple ACME issuer could be defined as:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: letsencrypt-prod
  namespace: edge-services
spec:
  acme:
    # The ACME server URL
    server: https://acme-v02.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: user@example.com
    # Name of a secret used to store the ACME account private key
    privateKeySecretRef:
      name: letsencrypt-prod
    # Enable HTTP01 validations
    http01: {}
```

This is the simplest of ACME issuers - it specifies no DNS-01 challenge providers. HTTP-01 validation can be performed through using Ingress resources by enabling the HTTP-01 challenge mechanism (with the `http01: {}` field). More information on configuring ACME Issuers can be found *here*.

4.2.1 Namespacing

An Issuer is a namespaced resource, and it is not possible to issue certificates from an Issuer in a different namespace. This means you will need to create an Issuer in each namespace you wish to obtain Certificates in.

If you want to create a single issuer than can be consumed in multiple namespaces, you should consider creating a *ClusterIssuer* resource. This is almost identical to the Issuer resource, however is non-namespaced and so it can be used to issue Certificates across all namespaces.

4.2.2 Ambient Credentials

Some API clients are able to infer credentials to use from the environment they run within. Notably, this includes cloud instance-metadata stores and environment variables. In cert-manager, the term 'ambient credentials' refers to such credentials. They are always drawn from the environment of the 'cert-manager-controller' deployment.

4.2.2.1 Example Usage

If cert-manager is deployed in an environment with ambient AWS credentials, such as with a kube2iam role, the following ClusterIssuer would make use of those credentials to perform the ACME DNS01 challenge with route53.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: user@example.com
    privateKeySecretRef:
      name: letsencrypt-prod
    dns01:
      providers:
      - name: route53
        route53:
          region: us-east-1
```

It is important to note that the `route53` section does not specify any `accessKeyID` or `secretAccessKeySecretRef`. If either of these are specified, ambient credentials will not be used.

4.2.2.2 When are Ambient Credentials used

Ambient credentials are supported for the 'route53' ACME DNS01 challenge provider.

They will only be used if no credentials are supplied, even if the supplied credentials are invalid.

By default, ambient credentials may be used by ClusterIssuers, but not regular issuers. The `--issuer-ambient-credentials` and `--cluster-issuer-ambient-credentials=false` flags on cert-manager may be used to override this behavior.

Note that ambient credentials are disabled for regular Issuers by default to ensure unprivileged users who may create issuers cannot issue certificates using any credentials cert-manager incidentally has access to.

4.2.3 Supported Issuer types

cert-manager has been designed to support pluggable Issuer backends. The currently supported Issuer types are:

Name	Description
ACME	Supports obtaining certificates from an ACME server, validating with HTTP01 or DNS01
CA	Supports issuing certificates using a simple signing keypair, stored in a Secret in the Kubernetes API server
Vault	Supports issuing certificates using HashiCorp Vault.
Self signed	Supports issuing self signed certificates

Each Issuer resource is of one, and only one type. The type of an Issuer is inferred by which field it specifies in its spec, such as `spec.acme` for the ACME issuer, or `spec.ca` for the CA based issuer.

4.2.3.1 ACME Configuration

In order to use the ACME provider, there are a number of required fields. For your ACME issuer to support the various ACME challenge mechanisms, you may need to provide some additional configuration on your resource, such as configuring credentials for a DNS provider.

4.2.3.1.1 HTTP01 Challenge Provider

In order to allow HTTP01 challenges to be solved, we must enable the HTTP01 challenge provider on our Issuer resource.

This is done through setting the `http01` field on the `issuer.spec.acme` stanza. Cert-manager will then attempt to solve ACME HTTP-01 challenges by using Ingress resources

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: example-issuer
spec:
  acme:
    email: user@example.com
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: example-issuer-account-key
    http01: {}
```

In rare cases it might be not possible/desired to use NodePort as type for the http01 challenge response service, e.g. because of Kubernetes limit restrictions. To define which Kubernetes service type to use during challenge response specify the following http01 config

```
http01:
  # Valid values are ClusterIP and NodePort
  serviceType: ClusterIP
```

By default type NodePort will be used when you don't set http01 or when you set serviceType to an empty string. Normally there's no need to change this.

Note: Let's Encrypt does not support issuing wildcard certificates with HTTP-01 challenges. To issue wildcard certificates, you must use the DNS-01 challenge.

Todo: Write a full description of how HTTP01 challenge validation works

---

4.2.3.1.2 DNS01 Challenge Provider

The ACME issuer can also contain DNS provider configuration, which can be used by Certificates using this Issuer in order to validate DNS01 challenge requests:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: example-issuer
spec:
  acme:
    email: user@example.com
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: example-issuer-account-key
    dns01:
      providers:
      - name: prod-clouddns
        clouddns:
          project: my-project
          serviceAccountSecretRef:
            name: prod-clouddns-svc-acct-secret
            key: service-account.json
```

Each issuer can specify multiple different DNS01 challenge providers, and it is also possible to have multiple instances of the same DNS provider on a single Issuer (e.g. two clouddns accounts could be set, each with their own name).

Setting nameservers for DNS01 self check

Cert-manager will check the correct DNS records exist before attempting a DNS01 challenge. By default, the DNS servers for this check will be taken from `/etc/resolv.conf`. If this is not desired (for example with multiple authoritative nameservers or split-horizon DNS), the cert-manager controller provides the `--dns01-self-check-nameservers` flag, which allows overriding the default nameservers with a comma seperated list of custom nameservers.

Example usage:

```
--dns01-self-check-nameservers "8.8.8.8:53,1.1.1.1:53"
```

Supported DNS01 providers

A number of different DNS providers are supported for the ACME issuer. Below is a listing of available providers, their *.yaml* configurations, along with additional Kubernetes and provider specific notes regarding their usage.

ACME-DNS

```
acmedns:
  host: https://acme.example.com
  accountSecretRef:
    name: acme-dns
    key: acmedns.json
```

In general, clients to acme-dns perform registration on the users behalf and inform them of the CNAME entries they must create. This is not possible in cert-manager, it is a non-interactive system. Registration must be carried out beforehand and the resulting credentials JSON uploaded to the cluster as a secret. In this example, we use `curl` and the API endpoints directly. Information about setting up and configuring acme-dns is available on the acme-dns project page.

1. First, register with the acme-dns server, in this example, there is one running at "auth.example.com"

   `curl -X POST http://auth.example.com/register` will return a JSON with credentials for your registration:


```
{
  "username":"eabcdb41-d89f-4580-826f-3e62e9755ef2",
  "password":"pbAXVjlIOE01xbut7YnAbkhMQIkcwoHO0ek2j4Q0",
  "fulldomain":"d420c923-bbd7-4056-ab64-c3ca54c9b3cf.auth.example.com",
  "subdomain":"d420c923-bbd7-4056-ab64-c3ca54c9b3cf",
  "allowfrom":[]
}
```


   It is strongly recommended to restrict the update endpoint to the IP range of your pods. This is done at registration time as follows:

   `curl -X POST http://auth.example.com/register -H "Content-Type: application/json" --data '{"allowfrom": ["10.244.0.0/16"]}'`

   Make sure to update the `allowfrom` field to match your cluster configuration. The JSON will now look like


```
{
  "username":"eabcdb41-d89f-4580-826f-3e62e9755ef2",
  "password":"pbAXVjlIOE01xbut7YnAbkhMQIkcwoHO0ek2j4Q0",
  "fulldomain":"d420c923-bbd7-4056-ab64-c3ca54c9b3cf.auth.example.com",
  "subdomain":"d420c923-bbd7-4056-ab64-c3ca54c9b3cf",
  "allowfrom":["10.244.0.0/16"]
}
```


2. Save this JSON to a file with the key as your domain. You can specify multiple domains with the same credentials if you like. In our example, the returned credentials can be used to verify ownership of "example.com" and and "example.org".


```
{
  "example.com": {
    "username":"eabcdb41-d89f-4580-826f-3e62e9755ef2",
    "password":"pbAXVjlIOE01xbut7YnAbkhMQIkcwoHO0ek2j4Q0",
    "fulldomain":"d420c923-bbd7-4056-ab64-c3ca54c9b3cf.auth.example.com",
    "subdomain":"d420c923-bbd7-4056-ab64-c3ca54c9b3cf",
    "allowfrom":["10.244.0.0/16"]
  },
  "example.org": {
    "username":"eabcdb41-d89f-4580-826f-3e62e9755ef2",
    "password":"pbAXVjlIOE01xbut7YnAbkhMQIkcwoHO0ek2j4Q0",
    "fulldomain":"d420c923-bbd7-4056-ab64-c3ca54c9b3cf.auth.example.com",
    "subdomain":"d420c923-bbd7-4056-ab64-c3ca54c9b3cf",
    "allowfrom":["10.244.0.0/16"]
  }
}
```


3. Next update your primary DNS server with CNAME record that will tell the verifier how to locate the challenge TXT record. This is obtained from the "fulldomain" field in the registration:

   `_acme-challenge.example.com CNAME d420c923-bbd7-4056-ab64-c3ca54c9b3cf.auth.example.com _acme-challenge.example.org CNAME d420c923-bbd7-4056-ab64-c3ca54c9b3cf.auth.example.com`

   Note that the "name" of the record is always the "_acme-challenge" subdomain, and the "value" of the record matches exactly the "fulldomain" field from registration.

At verification time, the domain name `d420c923-bbd7-4056-ab64-c3ca54c9b3cf.auth.example.com` will be a TXT record that is set to your validation token. When the verifier queries `_acme-challenge.example.com`, it will be directed to the correct location by this CNAME record. This proves that you control "example.com"

4. Create a secret from the credentials json that was saved in step 2, this secret is referenced in the `accountSecretRef` field of your dns01 issuer settings.

   ```
   kubectl create secret generic acme-dns --from-file acmedns.json
   ```

Akamai FastDNS

```
akamai:
  serviceConsumerDomain: akab-tho6xie2aiteip8p-poith5aej0ughaba.luna.akamaiapis.net
  clientTokenSecretRef:
    name: akamai-dns
    key: clientToken
  clientSecretSecretRef:
    name: akamai-dns
    key: clientSecret
  accessTokenSecretRef:
    name: akamai-dns
    key: accessToken
```

AzureDNS

Configuring the AzureDNS DNS-01 Challenge for a Kubernetes cluster requires creating a service principal in Azure.

For security purposes, it is appropriate to utilize RBAC to ensure that you properly maintain access control to your resources in Azure. The service principal that is generated by this tutorial has fine grained access to ONLY the DNS Zone in the specific resource group specified. It requires this permission so that it can read/write the _acme_challenge TXT records to the zone.

To create the service principal:

```
AZURE_CERT_MANAGER_SP_NAME=SOME_SERVICE_PRINCIPAL_NAME
AZURE_CERT_MANAGER_SP_PASSWORD=SOME_PASSWORD
AZURE_CERT_MANAGER_DNS_RESOURCE_GROUP=SOME_RESOURCE_GROUP
AZURE_CERT_MANAGER_DNS_NAME=SOME_DNS_ZONE

AZURE_CERT_MANAGER_SP_APP_ID=$(az ad sp create-for-rbac --name $AZURE_CERT_MANAGER_SP_
↪NAME --password $AZURE_CERT_MANAGER_SP_PASSWORD --query "appId" --output tsv)

# Lower the Permissions of the SP
az role assignment delete --assignee $AZURE_CERT_MANAGER_SP_APP_ID --role Contributor

# Give Access to DNS Zone
DNS_ID=$(az network dns zone show --name $AZURE_CERT_MANAGER_DNS_NAME --resource-
↪group $AZURE_CERT_MANAGER_DNS_RESOURCE_GROUP --query "id" --output tsv)

az role assignment create --assignee $AZURE_CERT_MANAGER_SP_APP_ID --role "DNS Zone␣
↪Contributor" --scope $DNS_ID

# Check Permissions
az role assignment list --assignee $AZURE_CERT_MANAGER_SP_APP_ID
```

(continues on next page)

(continued from previous page)

```

# Create Secret
kubectl create secret generic azuredns-config \
  --from-literal=CLIENT_SECRET=$AZURE_CERT_MANAGER_SP_PASSWORD

# Get the Service Principal App ID for configuration
echo $AZURE_CERT_MANAGER_SP_APP_ID
```

You can configure the issuer like so:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: example@example.com
    privateKeySecretRef:
      name: letsencrypt-prod
    dns01:
        providers:
            - name: azure
              azuredns:
                # Service principal clientId (also called appId)
                clientID: AZURE_SERVICE_PRINCIPAL_ID
                # A secretKeyRef to a service principal ClientSecret (password)
                # ref: https://docs.microsoft.com/en-us/azure/container-service/
↪kubernetes/container-service-kubernetes-service-principal
                clientSecretSecretRef:
                  name: AZUREDNS_SECRET_KEY_NAME
                  key: CLIENT_SECRET
                # Azure subscription Id
                subscriptionID: AZURE_SUBSCRIPTION_ID
                # Azure AD tenant Id
                tenantID: AZURE_TENANT_ID
                # ResourceGroup name where dns zone is provisioned
                resourceGroupName: AZURE_RESOURCE_GROUP
                hostedZoneName: AZURE_DNS_ZONE_NAME
```

Cloudflare

```
cloudflare:
  email: my-cloudflare-acc@example.com
  apiKeySecretRef:
    name: cloudflare-api-key-secret
    key: api-key
```

Google CloudDNS

```
clouddns:
  project: my-project
```

(continues on next page)

(continued from previous page)

```
  serviceAccountSecretRef:
    name: prod-clouddns-svc-acct-secret
    key: service-account.json
```

Amazon Route53

```
route53:
  region: eu-west-1

  # optional if ambient credentials are available; see ambient credentials␣
↪documentation
  accessKeyID: AKIAIOSFODNN7EXAMPLE
  secretAccessKeySecretRef:
    name: prod-route53-credentials-secret
    key: secret-access-key
```

Cert-manager requires the following IAM policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "route53:GetChange",
            "Resource": "arn:aws:route53:::change/*"
        },
        {
            "Effect": "Allow",
            "Action": "route53:ChangeResourceRecordSets",
            "Resource": "arn:aws:route53:::hostedzone/*"
        },
        {
            "Effect": "Allow",
            "Action": "route53:ListHostedZonesByName",
            "Resource": "*"
        }
    ]
}
```

The `route53:ListHostedZonesByName` statement can be removed if you specify the optional hosted zone ID (`spec.acme.dns01.providers[].hostedZoneID`) on the Issuer resource. You can further tighten this policy by limiting the hosted zone that cert-manager has access to (replace `arn:aws:route53:::hostedzone/*` with `arn:aws:route53:::hostedzone/DIKER8JPL21PSA`, for instance).

DigitalOcean

This provider uses a Kubernetes `Secret` Resource to work. In the following example, the secret will have to be named `digitalocean-dns` and have a subkey `access-token` with the token in it.

To create a Personnal Access Token, see DigitalOcean documentation. Handy direct link: https://cloud.digitalocean.com/account/api/tokens/new

```
digitalocean:
  tokenSecretRef:
    name: digitalocean-dns
    key: access-token
```

4.2.3.2 CA Configuration

CA Issuers issue certificates signed from a X509 signing keypair, stored in a secret in the Kubernetes API server.

You can find user guides on using the CA Issuer in the *CA Issuer tutorials section*.

Todo: Expand out CA Issuer reference documentation

4.2.3.3 Vault Configuration

Vault Issuers issue certificates from Hashicorp's Vault.

You can find user guides on using the Vault Issuer in the *Vault Issuer tutorials section*.

Todo: Expand out Vault Issuer reference documentation

4.2.3.4 Self-signed Configuration

Self signed Issuers will issue self signed certificates.

This is useful when building PKI within Kubernetes, or as a means to generate a root CA for use with the *CA Issuer* once cert-manager supports setting the `isCA` flag on Certificate resources (#85).

A self-signed Issuer contains no additional configuration fields, and can be created with a resource like so:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: selfsigning-issuer
spec:
  selfSigned: {}
```

Note: The presence of the `selfSigned: {}` line is enough to indicate that this Issuer is of type 'self signed'.

Once created, you should be able to issue certificates like usual by referencing the newly created Issuer in your `issuerRef`:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example-crt
spec:
  secretName: my-selfsigned-cert
  dnsNames:
```

(continues on next page)

(continued from previous page)

```
  - example.com
  issuerRef:
    name: selfsigning-issuer
    kind: ClusterIssuer
```

4.3 ClusterIssuers

ClusterIssuers are a resource type similar to *Issuers*. They are specified in exactly the same way, but they do not belong to a single namespace and can be referenced by Certificate resources from multiple different namespaces.

They are particularly useful when you want to provide the ability to obtain certificates from a central authority (e.g. Letsencrypt, or your internal CA) and you run single-tenant clusters.

The docs for Issuer resources apply equally to ClusterIssuers.

You can specify a ClusterIssuer resource by changing the `kind` attribute of an Issuer to `ClusterIssuer`, and removing the `metadata.namespace` attribute:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
...
```

We can then reference a ClusterIssuer from a Certificate resource by setting the `spec.issuerRef.kind` field to ClusterIssuer:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: my-certificate
  namespace: my-namespace
spec:
  secretName: my-certificate-secret
  issuerRef:
    name: letsencrypt-prod
    kind: ClusterIssuer
  ...
```

When referencing a `Secret` resource in `ClusterIssuer` resources (eg `apiKeySecretRef`) the `Secret` needs to be in the same namespace as the `cert-manager` controller pod. You can optionally override this by using the `--cluster-resource-namespace` argument to the controller.

For more information on configuring Issuer resources, see the *Issuers* reference documentation.

4.4 ingress-shim

cert-manager can be configured to automatically provision TLS certificates for Ingress resources via annotations on your Ingresses.

A small sub-component of cert-manager, ingress-shim, is responsible for this.

4.4.1 How it works

ingress-shim watches Ingress resources across your cluster. If it observes an Ingress with *any* of the annotations described in the 'Usage' section, it will ensure a Certificate resource with the same name as the Ingress, and configured as described on the Ingress exists. For example:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    # add an annotation indicating the issuer to use.
    certmanager.k8s.io/cluster-issuer: nameOfClusterIssuer
  name: myIngress
  namespace: myIngress
spec:
  rules:
  - host: myingress.com
    http:
      paths:
      - backend:
          serviceName: myservice
          servicePort: 80
        path: /
  tls: # < placing a host in the TLS config will indicate a cert should be created
  - hosts:
    - myingress.com
    secretName: myingress-cert # < cert-manager will store the created certificate in␣
↪this secret.
```

4.4.2 Configuration

Since cert-manager v0.2.2, ingress-shim is deployed automatically as part of a Helm chart installation.

If you would also like to use the old kube-lego `kubernetes.io/tls-acme: "true"` annotation for fully automated TLS, you will need to configure a default Issuer when deploying cert-manager. This can be done by adding the following `--set` when deploying using Helm:

```
--set ingressShim.defaultIssuerName=letsencrypt-prod \
--set ingressShim.defaultIssuerKind=ClusterIssuer
```

In the above example, cert-manager will create Certificate resources that reference the ClusterIssuer *letsencrypt-prod* for all Ingresses that have a `kubernetes.io/tls-acme: "true"` annotation.

For more information on deploying cert-manager, read the *deployment guide*.

4.4.3 Supported annotations

You can specify the following annotations on ingresses in order to trigger Certificate resources to be automatically created:

- `certmanager.k8s.io/issuer` - the name of an Issuer to acquire the certificate required for this ingress from. The Issuer **must** be in the same namespace as the Ingress resource.
- `certmanager.k8s.io/cluster-issuer` - the name of a ClusterIssuer to acquire the certificate required for this ingress from. It does not matter which namespace your Ingress resides, as ClusterIssuers are non-namespaced resources.

- `certmanager.k8s.io/acme-challenge-type` - by default, if the Issuer specified is an ACME issuer (either through ingress-shim's defaults, or with one of the above annotations), the ingress-shim will set the ACME challenge mechanism on the Certificate resource it creates to 'http01'. This annotation can be used to alter this behaviour. Must be one of 'http01' or 'dns01'.
- `certmanager.k8s.io/acme-dns01-provider` - if the ACME challenge type has been set to dns01, this annotation **must** be specified to instruct cert-manager which DNS provider (as configured on the specified Issuer resource) should be used. This field is required if the challenge type is set to DNS01.
- `certmanager.k8s.io/acme-http01-ingress-class` - if the ACME challenge type has been set to http01, this annotation allows you to configure ingress class that will be used to solve challenges for this ingress. Customising this is useful when you are trying to secure internal services, and need to solve challenges using different ingress class to that of the ingress. If not specified and the 'acme-http01-edit-in-place' annotation is not set, this defaults to the ingress class of the ingress resource.
- `kubernetes.io/tls-acme: "true"` - this annotation requires additional configuration of the ingress-shim (see above). Namely, a default issuer must be specified as arguments to the ingress-shim container.
- `certmanager.k8s.io/acme-http01-edit-in-place: "true"` - if the ACME challenge type has been set to http01, and the ingress has the 'kubernetes.io/tls-acme: true' annotation, this controls whether the ingress is modified 'in-place', or a new one created specifically for the http01 challenge. If present, and set to "true" the existing ingress will be modified. Any other value, or the absence of the annotation assumes "false".

4.5 API documentation

CHAPTER 5

Development documentation

5.1 Develop with minikube

Minikube is a tool to quickly provision a local Kubernetes cluster on many platforms. It can be used to test and develop cert-manager. This guide will walk you through getting started using Minikube for development.

5.1.1 Start minikube

First, run minikube, and configure your local kubectl command to work with minikube; minikube typically does this automatically.

```
# Check your locally installed minikube version
$ minikube version
minikube version: v0.25.0

# Start a local cluster
$ minikube start --extra-config=apiserver.Authorization.Mode=RBAC

# Verify it works. This should output a local apiserver IP
$ kubectl cluster-info

# Create a cluster role binding so Tiller has cluster-admin access rights
$ kubectl create clusterrolebinding default-admin --clusterrole=cluster-admin --
↪serviceaccount=kube-system:default

# Install helm
$ helm init
```

5.1.2 Install local development tools

You will need the following tools to build cert-manager:

- Bazel
- Docker (and enable for non-root user)

These instructions have only been tested on Linux; Windows and MacOS may require further changes.

If you need to add dependencies, you will additionally need:

- Git
- Mercurial

You can then run `bazel run //hack:update-deps` to regenerate any dependencies, and `bazel build :images` to build the docker images.

5.1.3 Build a dev version of cert-manager

```
# Configure your local docker client to use the minikube docker daemon
$ eval "$(minikube docker-env)"

# Build cert-manager binaries and docker images. Full output omitted for brevity
$ make build
Successfully tagged quay.io/jetstack/cert-manager-controller:build
```

5.1.4 Deploy that version with helm

```
# Install our freshly built cert-manager image
$ helm install \
     --set image.tag=build \
     --set image.pullPolicy=Never \
     --name cert-manager \
     ./contrib/charts/cert-manager
```

From here, you should be able to do whatever manual testing or development you wish to.

5.1.5 Deploy a new version

In general, upgrading can be done simply by running *make build*, and then deleting the deployed pod using *kubectl delete pod*.

However, if you make changes to the helm chart or wish to change the controller's arguments, such as to change the logging level, you may also update it with the following:

```
helm upgrade  \
     cert-manager \
     --reuse-values \
     --set extraArgs="{-v=5}"
     --set image.tag=build
     ./contrib/charts/cert-manager
```

5.2 Running end-to-end tests

cert-manager has an end-to-end test suite that verifies functionality against a real Kubernetes cluster.

This document explains how you can run the end-to-end tests yourself. This is useful when you have added or changed functionality in cert-manager and want to verify the software still works as expected.

5.2.1 Requirements

Currently, a number of tools **must** be installed on your machine in order to run the tests:

- `bazel` - As with all other development, Bazel is required to actually build the project as well as end-to-end test framework. Bazel will also retrieve appropriate versions of any other dependencies depending on what 'target' you choose to run.
- `docker` - We provision a whole Kubernetes cluster within Docker, and so an up to date version of Docker must be installed. The oldest Docker version we have tested is 17.09.
- `kubectl` - If you are running the tests on Linux, this step is technically not required. For non-Linux hosts (i.e. OSX), you will need to ensure you have a relatively new version of kubectl available on your PATH.
- An internet connection - tests require access to DNS, and optionally Cloudflare APIs (if a Cloudflare API token is provided).

Bazel, Docker and Kubectl should be installed through your preferred means.

5.2.2 Run end-to-end tests

You can run the end-to-end tests by executing the following:

```
./hack/ci/run-e2e-kind.sh
```

The full suite may take up to 10 minutes to run. You can monitor output of this command to track progress.

5.3 Contributing DNS01 providers

Steps to add a `FooDNS` DNS-01 provider:

1. Create a new package under `pkg/issuer/acme/dns/foodns`. This is where all the code to interact with the DNS providers API will live.
2. Implement functions to match the solver interface (`Present`, `CleanUp` and `Timeout`). Use an existing provider for reference. Most of the cert-manager providers are based off https://github.com/xenolf/lego, so if lego supports the DNS provider you want to add, it's fairly easy to copy it over and make modifications to fit with the cert-manager codebase. Examples of the changes required:
   - replace uses of `github.com/xenolf/lego/acme` with `github.com/jetstack/cert-manager/pkg/issuer/acme/dns/util`.
   - replace uses of `github.com/xenolf/lego/log` with `github.com/golang/glog`.
   - remove references to `github.com/xenolf/lego/platform/config/env`. cert-manager does not use environment variables for internal configuration, so calls to this package should not be required.
3. Add unit test coverage for this package.
4. Add your provider configuration types to the API (located in `pkg/apis/certmanager/v1alpha1/types.go`) and regenerate code (run `./hack/update-codegen.sh`). New API types should have an associated short documentation string, which is added to the reference API documentation (run `./hack/update-reference-docs-dockerized.sh` to update the API documentation).

5. Register the provider in `pkg/issuer/acme/dns`:
    - The constructor for the provider needs adding to `dnsProviderConstructors`,
    - `solverForIssuerProvider` must be updated to handle retrieving any information for the new provider (for example, fetching credentials from a secret) and constructing a new instance of the provider.
6. Add coverage for the provider to `pkg/issuer/acme/dns/dns_test.go`.
7. Add example configuration for the new provider to `docs/reference/issuers/acme/dns01/index.rst`. The more information here the better, this example and corresponding documentation should inform users how to use and configure this backend, as well as mentioning any nuances with using this particular provider.
8. Test your provider out against a real account, and make sure you can issue a Certificate.
9. Submit your new provider to cert-manager!

Things to watch out for:

- Assume that at any point the cert-manager process may restart. Make sure values required for operations like `CleanUp` are not solely stored in memory.

5.4 DCO Sign off

All authors to the project retain copyright to their work. However, to ensure that they are only submitting work that they have rights to, we are requiring everyone to acknowledge this by signing their work.

Any copyright notices in this repo should specify the authors as "the Jetstack cert-manager contributors".

To sign your work, just add a line like this at the end of your commit message:

```
Signed-off-by: Joe Bloggs <joe@example.com>
```

This can easily be done with the `--signoff` option to `git commit`. You can also mass sign-off a whole PR with `git rebase --signoff master`, replacing `master` with the branch you are creating a pull request again if not master.

By doing this you state that you certify the following (from https://developercertificate.org/):

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
1 Letterman Drive
Suite D4700
San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.


Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or
```

(continues on next page)

(continued from previous page)

```
(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

5.5 Release process

This document aims to outline the process that should be followed for cutting a new release of cert-manager.

5.5.1 Minor releases

A minor release is a backwards-compatible 'feature' release. It can contain new features and bugfixes.

5.5.1.1 Release schedule

We aim to cut a new minor release once per month. The rough goals for each release are outlined as part of a GitHub milestone. We cut a release even if some of these goals are missed, in order to keep up release velocity.

5.5.1.2 Process

Note: This process document is WIP and may be incomplete

The process for cutting a minor release is as follows:

1. Ensure upgrading document exists in docs/admin/upgrading
2. Create a new release branch (e.g. `release-0.5`)
3. Push it to the `jetstack/cert-manager` repository
4. Create a pull-request updating the Helm chart version and merge it:
    - Update contrib/charts/cert-manager/README.md
    - Update contrib/charts/cert-manager/Chart.yaml
    - Update contrib/charts/cert-manager/values.yaml
    - Update contrib/charts/cert-manager/requirements.yaml

- Update contrib/charts/cert-manager/webhook/Chart.yaml
- Update contrib/charts/cert-manager/webhook/values.yaml
- Run `helm dep update` in the contrib/charts/cert-manager directory
- Run `./hack/update-deploy-gen.sh` in the root of the repository

5. Gather release notes since the previous release:
   - Run `relnotes -repo cert-manager -owner jetstack release-0.5`
   - Write up appropriate notes, similar to previous releases
6. Submit the Helm chart changes to the upstream `helm/charts` repo:

```
TARGET_REPO_REMOTE=upstream \
SOURCE_REPO_REMOTE=upstream \
SOURCE_REPO_REF=release-0.5 \
GITHUB_USER=munnerz \
./hack/create-chart-pr.sh
```

7. Iterate on review feedback (hopefully this will be minimal) and submit changes to `master` of cert-manager, performing a rebase of release-x.y and re-run of the `create-chart-pr.sh` script after each cycle to gather more feedback.
8. Create a new tag taken from the release branch, e.g. `v0.5.0`.

5.5.2 Patch releases

A patch release contains critical bugfixes for the project. They are managed on an ad-hoc basis, and should only be required when critical bugs/regressions are found in the release.

We will only perform patch release for the **current** version of cert-manager.

Once a new minor release has been cut, we will stop providing patches for the version before it.

5.5.2.1 Release schedule

Patch releases are cut on an ad-hoc basis, depending on recent activity on the release branch.

5.5.2.2 Process

Note: This process document is WIP and may be incomplete

Bugs that need to be fixed in a patch release should be cherry picked into the appropriate release branch using the `./hack/cherry-pick-pr.sh` script in this repository.

The process for cutting a patch release is as follows:

1. Create a PR against the **release branch** to bump the chart version:
   - Update contrib/charts/cert-manager/README.md
   - Update contrib/charts/cert-manager/Chart.yaml
   - Update contrib/charts/cert-manager/values.yaml
   - Update contrib/charts/cert-manager/requirements.yaml

- Update contrib/charts/cert-manager/webhook/Chart.yaml
- Update contrib/charts/cert-manager/webhook/values.yaml
- Run `helm dep update` in the contrib/charts/cert-manager directory
- Run `./hack/update-deploy-gen.sh` in the root of the repository

2. Submit the Helm chart changes to the upstream `helm/charts` repo:

```
TARGET_REPO_REMOTE=upstream \
SOURCE_REPO_REMOTE=upstream \
SOURCE_REPO_REF=release-0.5 \
GITHUB_USER=munnerz \
./hack/create-chart-pr.sh
```

3. Iterate on review feedback (hopefully this will be minimal) and submit changes to `master` of cert-manager, performing a rebase of release-x.y and re-run of the `create-chart-pr.sh` script after each cycle to gather more feedback.
4. Gather release notes since the previous release:
   - Run `relnotes -repo cert-manager -owner jetstack release-0.5`
   - Write up appropriate notes, similar to previous patch releases
5. Create a new tag taken from the release branch, e.g. `v0.5.1`.

5.6 Generating Documentation

The documentation is generated from reStructured Text by Sphinx (via Read The Docs). If you're unfamiliar with reStructured Text, the files typically have the extension *.rst*. You can find more details in the reStructured Text Basics.

5.6.1 Installation instructions

To install the sphinx tools, you'll need `python` (and `pip`) installed.:

```
.. code-block: shell
```

pip install –user -r requirements.txt

5.6.2 Generating documentation locally

You can generate the documentation locally with the following command:

This will create documentation in the `_build` directory which you can open with your browser.

Note that you do not need to add these files to your git client, as *Read The Docs* will generate the HTML on the fly.