cert-manager Documentation

Jetstack Ltd

Sep 27, 2019

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, Venafi, 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

Get started

The guides in this section will explain how to install and set up cert-manager.

If you run into issues deploying, upgrading or running cert-manager please check the *troubleshooting* document.

1.1 Installing cert-manager

cert-manager supports running on Kubernetes and OpenShift. The installation mechanism between the two platforms is similar, although there are a number of extra notes to be aware of per-platform.

1.1.1 Installing on Kubernetes

cert-manager runs within your Kubernetes cluster as a series of deployment resources. It utilises CustomResourceDefinitions to configure Certificate Authorities and request certificates.

It is deployed using regular YAML manifests, like any other applications on Kubernetes.

Once cert-manager has been deployed, you must configure Issuer or ClusterIssuer resources which represent certificate authorities. More information on configuring different Issuer types can be found in the *respective setup guides*.

> **Warning:** You should not install multiple instances of cert-manager on a single cluster. This will lead to undefined behaviour and you may be banned from providers such as Let's Encrypt.

Installing with regular manifests

In order to install cert-manager, we must first create a namespace to run it within. This guide will install cert-manager into the `cert-manager` namespace. It is possible to run cert-manager in a different namespace, although you will need to make modifications to the deployment manifests.

```
# Create a namespace to run cert-manager in
kubectl create namespace cert-manager
```

As part of the installation, cert-manager also deploys a ValidatingWebhookConfiguration resource in order to validate that the Issuer, ClusterIssuer and Certificate resources we will create after installation are valid.

In order to deploy the ValidatingWebhookConfiguration, cert-manager creates a number of 'internal' Issuer and Certificate resources in its own namespace.

This creates a chicken-and-egg problem, where cert-manager requires the webhook in order to create the resources, and the webhook requires cert-manager in order to run.

We avoid this problem by disabling resource validation on the namespace that cert-manager runs in:

```
# Disable resource validation on the cert-manager namespace
kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true
```

You can read more about the webhook on the *webhook document*.

We can now go ahead and install cert-manager. All resources (the CustomResourceDefinitions, cert-manager, and the webhook component) are included in a single YAML manifest file:

```
# Install the CustomResourceDefinitions and cert-manager itself
kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/v0.10.1/
↪cert-manager.yaml
```

Note: If you are running kubectl v1.12 or below, you will need to add the `--validate=false` flag to your `kubectl apply` command above else you will receive a validation error relating to the `caBundle` field of the `ValidatingWebhookConfiguration` resource. This issue is resolved in Kubernetes 1.13 onwards. More details can be found in kubernetes/kubernetes#69590.

Note: When running on GKE (Google Kubernetes Engine), you may encounter a 'permission denied' error when creating some of these resources. This is a nuance of the way GKE handles RBAC and IAM permissions, and as such you should 'elevate' your own privileges to that of a 'cluster-admin' **before** running the above command. If you have already run the above command, you should run them again after elevating your permissions:

```
kubectl create clusterrolebinding cluster-admin-binding \
  --clusterrole=cluster-admin \
  --user=$(gcloud config get-value core/account)
```

Installing with Helm

As an alternative to the YAML manifests referenced above, we also provide an official Helm chart for installing cert-manager.

Pre-requisites

- Helm and Tiller installed (or alternatively, use Tillerless Helm v2)
- cluster-admin privileges bound to the Tiller pod

Foreword

Before deploying cert-manager with Helm, 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.

Steps

In order to install the Helm chart, you must run:

```
# Install the CustomResourceDefinition resources separately
kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/release-0.10/
↪deploy/manifests/00-crds.yaml

# Create the namespace for cert-manager
kubectl create namespace cert-manager

# Label the cert-manager namespace to disable resource validation
kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true

# Add the Jetstack Helm repository
helm repo add jetstack https://charts.jetstack.io

# Update your local Helm chart repository cache
helm repo update

# Install the cert-manager Helm chart
helm install \
  --name cert-manager \
  --namespace cert-manager \
  --version v0.10.1 \
  jetstack/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.

Verifying the installation

Once you've installed cert-manager, you can verify it is deployed correctly by checking the `cert-manager` namespace for running pods:

```
kubectl get pods --namespace cert-manager

NAME                                       READY   STATUS    RESTARTS   AGE
cert-manager-5c6866597-zw7kh               1/1     Running   0          2m
cert-manager-cainjector-577f6d9fd7-tr77l   1/1     Running   0          2m
cert-manager-webhook-787858fcdb-nlzsq      1/1     Running   0          2m
```

You should see the `cert-manager`, `cert-manager-cainjector` and `cert-manager-webhook` pod in a Running state. It may take a minute or so for the TLS assets required for the webhook to function to be provisioned. This may cause the webhook to take a while longer to start for the first time than other pods. If you experience problems, please check the *troubleshooting guide*.

The following steps will confirm that cert-manager is set up correctly and able to issue basic certificate types:

```
# Create a ClusterIssuer to test the webhook works okay
cat <<EOF > test-resources.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: cert-manager-test
---
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: test-selfsigned
  namespace: cert-manager-test
spec:
  selfSigned: {}
---
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: selfsigned-cert
  namespace: cert-manager-test
spec:
  commonName: example.com
  secretName: selfsigned-cert-tls
  issuerRef:
    name: test-selfsigned
EOF

# Create the test resources
kubectl apply -f test-resources.yaml

# Check the status of the newly created certificate
# You may need to wait a few seconds before cert-manager processes the
# certificate request
kubectl describe certificate -n cert-manager-test
...
Spec:
  Common Name:  example.com
  Issuer Ref:
    Name:       test-selfsigned
  Secret Name:  selfsigned-cert-tls
Status:
  Conditions:
    Last Transition Time:  2019-01-29T17:34:30Z
    Message:               Certificate is up to date and has not expired
    Reason:                Ready
    Status:                True
    Type:                  Ready
  Not After:               2019-04-29T17:34:29Z
Events:
  Type    Reason      Age   From          Message
  ----    ------      ----  ----          -------
```

(continues on next page)

(continued from previous page)

```
  Normal  CertIssued  4s    cert-manager  Certificate issued successfully

# Clean up the test resources
kubectl delete -f test-resources.yaml
```

If all the above steps have completed without error, you are good to go!

If you experience problems, please check the *troubleshooting guide*.

Configuring your first Issuer

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

You should read the *Setting up Issuers* guide to learn how to configure cert-manager to issue certificates from one of the supported backends.

Alternative installation methods

Helmfile

Helmfile is a declarative spec for deploying helm charts.

'cert-manager-installer': https://github.com/zakkg3/cert-manager-installer It's an easy and automated way to install cert-manager.

Note: This is an external link and it's not officially maintained by cert-manager but by the community.

```
git clone git@github.com:zakkg3/cert-manager-installer.git
cd cert-manager-installer
helmfile sync
```

kubeprod

Bitnami Kubernetes Production Runtime (BKPR, `kubeprod`) is a curated collection of the services you would need to deploy on top of your Kubernetes cluster to enable logging, monitoring, certificate management, automatic discovery of Kubernetes resources via public DNS servers and other common infrastructure needs.

It depends on `cert-manager` for certificate management, and it is regularly tested so the components are known to work together for GKE and AKS clusters (EKS to be added soon). For its ingress stack it creates a DNS entry in the configured DNS zone and requests a TLS certificate from the Let's Encrypt staging server.

BKPR can be deployed using the `kubeprod install` command, which will deploy `cert-manager` as part of it. Details available in the BKPR installation guide.

Debugging installation issues

If you have any issues with your installation, please refer to the *troubleshooting guide*.

1.1.2 Installing on OpenShift

cert-manager supports running on OpenShift in a similar manner to *Running on Kubernetes*. It runs within your OpenShift cluster as a series of deployment resources. It utilises CustomResourceDefinitions to configure Certificate Authorities and request certificates.

It is deployed using regular YAML manifests, like any other application on OpenShift.

Once cert-manager has been deployed, you must configure Issuer or ClusterIssuer resources which represent certificate authorities. More information on configuring different Issuer types can be found in the *respective setup guides*.

> **Warning:** You should not install multiple instances of cert-manager on a single cluster. This will lead to undefined behaviour and you may be banned from providers such as Let's Encrypt.

Login to your OpenShift cluster

Before you can install cert-manager, you must first ensure your local machine is configured to talk to your OpenShift cluster using the `oc` tool.

```
# Login to the OpenShift cluster as the system:admin user
oc login -u system:admin
```

Installing with regular manifests

In order to install cert-manager, we must first create a namespace to run it within. This guide will install cert-manager into the `cert-manager` namespace. It is possible to run cert-manager in a different namespace, although you will need to make modifications to the deployment manifests.

```
# Create a namespace to run cert-manager in
oc create namespace cert-manager
```

As part of the installation, cert-manager also deploys a ValidatingWebhookConfiguration resource in order to validate that the Issuer, ClusterIssuer and Certificate resources we will create after installation are valid.

In order to deploy the ValidatingWebhookConfiguration, cert-manager creates a number of 'internal' Issuer and Certificate resources in its own namespace.

This creates a chicken-and-egg problem, where cert-manager requires the webhook in order to create the resources, and the webhook requires cert-manager in order to run.

We avoid this problem by disabling resource validation on the namespace that cert-manager runs in:

```
# Disable resource validation on the cert-manager namespace
oc label namespace cert-manager certmanager.k8s.io/disable-validation=true
```

You can read more about the webhook on the *webhook document*.

We can now go ahead and install cert-manager. All resources (the CustomResourceDefinitions, cert-manager, and the webhook component) are included in a single YAML manifest file:

```
# Install the CustomResourceDefinitions and cert-manager itself
oc apply --validate=false -f https://github.com/jetstack/cert-manager/releases/
↪download/v0.10.1/cert-manager-openshift.yaml
```

Note: The `--validate=false` flag is added to the `oc apply` command above else you will receive a validation error relating to the `caBundle` field of the `ValidatingWebhookConfiguration` resource.

Configuring your first Issuer

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

You should read the *Setting up Issuers* guide to learn how to configure cert-manager to issue certificates from one of the supported backends.

Debugging installation issues

If you have any issues with your installation, please refer to the *troubleshooting guide*.

1.2 Webhook component

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

This allows cert-manager to validate that Issuer, ClusterIssuer and Certificate resources that are submitted to the apiserver are syntactically valid, and catch issues with your resources early on.

If you disable the webhook component, cert-manager will still perform the same resource validation however it will not reject 'create' events when the resources are submitted to the apiserver if they are invalid. This means it may be possible for a user to submit a resource that renders the controller inoperable. For this reason, it is strongly advised to keep the webhook **enabled**.

Note: This feature requires Kubernetes v1.9 or greater.

1.2.1 How it works

This sections walks through how the resource validation webhook is configured and explains the process required for it to provision.

The webhook is a ValidatingWebhookConfiguration resource combined with an extra pod that is deployed alongside the cert-manager-controller.

The ValidatingWebhookConfiguration instructs the Kubernetes apiserver to POST the contents of any Create or Update operations performed on cert-manager resource types in order to validate that they are setting valid configurations.

This allows us to ensure mis-configurations are caught early on and communicated to you.

In order for this to work, the webhook requires a TLS certificate that the apiserver is configured to trust.

The cert-manager deployment manifests define two Issuer resources, and two Certificate resources:

- issuer/cert-manager-webhook-selfsign - A self signing Issuer that is used to issue a self signed root CA certificate.
- certificate/cert-manager-webhook-ca - A self-signed root CA certificate which is used to sign certificates for the webhook pod.

- issuer/cert-manager-webhook-ca - A CA Issuer that is used to issue certificates used by the webhook pod to serve with.
- certificate/cert-manager-webhook-webhook-tls - A TLS certificate issued by the root CA above, served by the webhook.

You can check the status of these resources to ensure they're functioning correctly by running:

```
kubectl get issuer --namespace cert-manager
NAME                              AGE
cert-manager-webhook-ca           10m
cert-manager-webhook-selfsign     10m

kubectl get certificate -o wide --namespace cert-manager
NAME                                  READY   SECRET                                  ISSUER␣
↪                           STATUS                                          AGE
cert-manager-webhook-ca               True    cert-manager-webhook-ca                 cert-
↪manager-webhook-selfsign   Certificate is up to date and has not expired   10m
cert-manager-webhook-webhook-tls      True    cert-manager-webhook-webhook-tls        cert-
↪manager-webhook-ca         Certificate is up to date and has not expired   10m
```

If the certificates or issuer are not Ready or you cannot see them, you should check the *troubleshooting* guide for help.

Note: If you are running Kubernetes v1.10 or earlier, you may need to run `kubectl describe` instead of `kubectl get` as the 'additionalPrinterColumns' functionality only moved to beta in v1.11.

cainjector

The *cert-manager CA injector* is responsible for injecting the two CA bundles above into the webhook's ValidatingWebhookConfiguration and APIService resource in order to allow the Kubernetes apiserver to 'trust' the webhook apiserver.

This component is configured using the `certmanager.k8s.io/inject-apiserver-ca: "true"` and `certmanager.k8s.io/inject-apiserver-ca: "true"` annotations on the APIService and ValidatingWebhookConfiguration resources.

It copies across the CA defined in the 'cert-manager-webhook-ca' Secret generated above to the `caBundle` field on the APIService resource. It also sets the webhook's `clientConfig.caBundle` field on the `cert-manager-webhook` ValidatingWebhookConfiguration resource to that of your Kubernetes API server in order to support Kubernetes versions earlier than v1.11.

Known issues

This section contains known issues with the webhook component.

If you're having problems, or receiving errors when creating cert-manager resources, please read through this section for help.

Disabling validation on the cert-manager namespace

If you've installed cert-manager with custom manifests, or have performed an upgrade from an earlier version, it's important to make sure that the namespace that the webhook is running in has an additional label applied to it in order to disable resource validation on the namespace that the webhook runs in.

If this step is not completed, cert-manager will not be able to provision certificates for the webhook correctly, causing a chicken-egg situation.

To apply the label, run:

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

You may need to wait a little while before cert-manager retries issuing the certificates if they have been failing for a while due to cert-manager's built in back-offs.

Running on private GKE clusters

When Google configure the control plane for private clusters, they automatically configure VPC peering between your Kubernetes cluster's network and a separate Google managed project.

In order to restrict what Google are able to access within your cluster, the firewall rules configured restrict access to your Kubernetes pods.

This means that in order to use the webhook component with a GKE private cluster, you must configure an additional firewall rule to allow the GKE control plane access to your webhook pod.

You can read more information on how to add firewall rules for the GKE control plane nodes in the GKE docs.

Alternatively, you can read how to *disable the webhook component* below.

Todo: add an example command for how to do this here & explain any security implications

---

1.2.2 Disable the webhook component

If you are having issues with the webhook and cannot use it at this time, you can optionally disable the webhook altogether.

Doing this may expose your cluster to mis-configuration problems that in some cases could cause cert-manager to stop working altogether (i.e. if invalid types are set for fields on cert-manager resources).

How you disable the webhook depends on your deployment method.

With Helm

The Helm chart exposes an option that can be used to disable the webhook.

To do so with an existing installation, you can run:

```
helm upgrade cert-manager \
   --reuse-values \
   --set webhook.enabled=false
```

If you have not installed cert-manager yet, you can add the `--set webhook.enabled=false` to the `helm install` command used to install cert-manager.

With static manifests

Because we cannot specify options when installing the static manifests to conditionally disable different components, we also ship a copy of the deployment files that do not include the webhook.

Instead of installing with cert-manager.yaml file, you should instead use the cert-manager-no-webhook.yaml file located in the deploy directory.

This is a destructive operation, as it will remove the CustomResourceDefinition resources, causing your configured Issuers, Certificates etc to be deleted.

You should first *backup your configuration* before running the following commands.

To re-install cert-manager without the webhook, run:

```
kubectl delete -f https://github.com/jetstack/cert-manager/releases/download/v0.10.1/
↪cert-manager.yaml

kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/v0.10.1/
↪cert-manager-no-webhook.yaml
```

Once you have re-installed cert-manager, you should then *restore your configuration*.

1.3 Troubleshooting installation

1.3.1 Internal error occurred: failed calling admission webhook ... the server is currently unable to handle the request

When installing or upgrading cert-manager, you may run into issues when going through the Validation Steps in the install guide which relate to the admission webhook.

If you see an error like the above, this guide will talk you through a few checks that can pick up common installation problems.

1. Check the namespace cert-manager is running in

As described in the *Webhook component* documentation, the webhook component requires TLS certificates in order to start and communicate securely with the Kubernetes API server.

In order for cert-manager to be able to issue certificates for the webhook before it has started, we must **disable** resource validation on the namespace that cert-manager is running in.

Assuming you have deployed into the `cert-manager` namespace, run the following command to verify that your cert-manager namespace has the necessary label:

```
kubectl describe namespace cert-manager

Name:         cert-manager
Labels:       certmanager.k8s.io/disable-validation=true
Annotations:  <none>
Status:       Active
...
```

If you cannot see the `certmanager.k8s.io/disable-validation=true` label on your namespace, you should add it with:

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

Please continue reading this guide once you have added the label.

2. Verify that the webhook Issuer and Certificate resources exist

If you had any issues upgrading, especially if you install cert-manager using Helm, you may run into an issue where either:

- the CustomResourceDefinition resources do not exist
- the webhook's Issuer and Certificate resources do not exist

We can first check for the existence of the CustomResourceDefinition resources:

```
kubectl get crd | grep certmanager

NAME                                          CREATED AT
certificates.certmanager.k8s.io               2018-08-17T20:12:26Z
challenges.certmanager.k8s.io                 2018-08-02T15:33:02Z
clusterissuers.certmanager.k8s.io             2018-08-17T20:12:26Z
issuers.certmanager.k8s.io                    2018-08-17T20:12:26Z
orders.certmanager.k8s.io                     2018-08-02T14:40:11Z
```

We should then also check for that the webhook's Issuer and Certificate resources exist and have been issued correctly:

```
kubectl get issuer,certificate --namespace cert-manager

NAME                                                        AGE
issuer.certmanager.k8s.io/cert-manager-webhook-ca           22d
issuer.certmanager.k8s.io/cert-manager-webhook-selfsign     22d

NAME                                                                READY   SECRET
↪                        AGE
certificate.certmanager.k8s.io/cert-manager-webhook-ca              True    cert-
↪manager-webhook-ca             22d
certificate.certmanager.k8s.io/cert-manager-webhook-webhook-tls     True    cert-
↪manager-webhook-webhook-tls    22d
```

If you do not see the CustomResourceDefinitions installed, or cannot see the webhook's Issuer and Certificate resources, please go back to the install guide and ensure you've followed every step closely.

Take particular care to install the CRD manifest **before** installing cert-manager itself.

3. Verify all cert-manager pods are running successfully

You can verify that cert-manager has managed to start successfully by checking the state of the pods that have been deployed:

```
kubectl get pods --namespace cert-manager

NAME                                       READY   STATUS      RESTARTS   AGE
cert-manager-7cbdc48784-rpgnt              1/1     Running     0          3m
cert-manager-webhook-5b5dd6999-kst4x       1/1     Running     0          3m
cert-manager-cainjector-3ba5cd2bcd-de332x  1/1     Running     0          3m
```

If the 'webhook' pod (2nd line) is in a ContainerCreating state, it may still be waiting for the Secret in step 2 to be mounted into the pod.

Provided the Secret resource **does** now exist, Waiting a few minutes, or deleting the pod and allowing it to be recreated should get things moving again.

Note: Check if the Secret exists by running:

```
kubectl --namespace cert-manager get secret cert-manager-webhook-webhook-tls
```

CHAPTER 2

Tutorials

This section contains guides that help you get started using cert-manager for more specific use cases.

For more information on performing individual tasks, read the *tasks section*.

2.1 ACME Issuer Tutorials

This sections contains tutorials relating to the ACME issuer.

2.1.1 Quick-Start using Cert-Manager with NGINX Ingress

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
```

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!
```

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
```

(continues on next page)

(continued from previous page)

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

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

==> 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
```

(continues on next page)

(continued from previous page)

```
              servicePort: 80
            path: /
    # This section is only required if TLS is to be enabled for the Ingress
    tls:
        - hosts:
            - www.example.com
          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.

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.

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/release-0.
↪10/docs/tutorials/acme/quick-start/example/deployment.yaml
deployment.extensions "kuard" created

$ kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/release-0.
↪10/docs/tutorials/acme/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:
```

(continues on next page)

(continued from previous page)

```
  name: kuard
  annotations:
    kubernetes.io/ingress.class: "nginx"
    #certmanager.k8s.io/issuer: "letsencrypt-staging"

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/
↪release-0.10/docs/tutorials/acme/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.

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 or plain Kubernetes manifest to install cert-manager.

Read the getting started guide to install cert-manager using your prefered method.

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.
>
> Note that if you're using this document as a guide to configure cert-manager for your own Issuer, you must create the Issuers in the same namespace as your Ingress resouces by adding '-n my-namespace' to your 'kubectl create' commands. Your other option is to replace your Issuers with ClusterIssuers. ClusterIssuer resources apply across all Ingress resources in your cluster and don't have this namespace-matching requirement.
>
> More information on the differences between Issuers and ClusterIssuers and when you might choose to use each can be found at:
>
> https://docs.cert-manager.io/en/latest/tasks/issuers/index.html#difference-between-issuers-and-clusterissuers

Certificate

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

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 Encrypt and used to notify you of certificate expirations and updates.

- staging issuer: staging-issuer.yaml

```
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
    solvers:
    - http01:
        ingress:
          class:  nginx
```

Once edited, apply the custom resource:

```
$ kubectl create --edit -f https://raw.githubusercontent.com/jetstack/cert-manager/
↪release-0.10/docs/tutorials/acme/quick-start/example/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.

- production issuer: production-issuer.yaml

```
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
    solvers:
    - http01:
        ingress:
          class: nginx
```

```
$ kubectl create --edit -f https://raw.githubusercontent.com/jetstack/cert-manager/
↪release-0.10/docs/tutorials/acme/quick-start/example/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:

```
$ kubectl describe issuer letsencrypt-staging

Name:         letsencrypt-staging
Namespace:    default
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration={"apiVersion":
↪"certmanager.k8s.io/v1alpha1","kind":"Issuer","metadata":{"annotations":{},"name":
↪"letsencrypt-staging","namespace":"default"},"spec":{"a...
API Version:  certmanager.k8s.io/v1alpha1
Kind:         Issuer
Metadata:
  Cluster Name:
  Creation Timestamp:  2018-11-17T18:03:54Z
  Generation:          0
  Resource Version:    9092
  Self Link:           /apis/certmanager.k8s.io/v1alpha1/namespaces/default/issuers/
↪letsencrypt-staging
  UID:                 25b7ae77-ea93-11e8-82f8-42010a8a00b5
Spec:
  Acme:
    Email:  your.email@your-domain.com
    Private Key Secret Ref:
      Key:
      Name:  letsencrypt-staging
    Server:  https://acme-staging-v02.api.letsencrypt.org/directory
    Solvers:
      Http 01:
        Ingress:
          Class:  nginx
Status:
  Acme:
    Uri:  https://acme-staging-v02.api.letsencrypt.org/acme/acct/7374163
  Conditions:
    Last Transition Time:  2018-11-17T18:04:00Z
    Message:               The ACME account was registered with the ACME server
    Reason:                ACMEAccountRegistered
    Status:                True
    Type:                  Ready
Events:                    <none>
```

You should see the issuer listed with a registered account.

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:

- ingress tls: ingress-tls.yaml

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

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

and apply it:

```
$ kubectl create --edit -f https://raw.githubusercontent.com/jetstack/cert-manager/
↪release-0.10/docs/tutorials/acme/quick-start/example/ingress-tls.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                      READY   SECRET                    AGE
quickstart-example-tls    True    quickstart-example-tls    16m
```

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
```

(continues on next page)

(continued from previous page)

```
    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:
  Dns Names:
    example.your-domain.com
  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
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
```

(continues on next page)

(continued from previous page)

```
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:

- ingress tls final: ingress-tls-final.yaml

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

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

```
$ kubectl create --edit -f https://raw.githubusercontent.com/jetstack/cert-manager/
↪release-0.10/docs/tutorials/acme/quick-start/example/ingress-tls-final.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
```

(continues on next page)

(continued from previous page)

```
  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
  Self Link:               /apis/certmanager.k8s.io/v1alpha1/namespaces/default/
↪certificates/quickstart-example-tls
  UID:                     bdd93b32-ea97-11e8-82f8-42010a8a00b5
Spec:
  Dns Names:
    example.your-domain.com
  Issuer Ref:
    Kind:       Issuer
    Name:       letsencrypt-prod
  Secret Name:  quickstart-example-tls
Status:
  Conditions:
    Last Transition Time:  2019-01-09T13:52:05Z
    Message:               Certificate does not exist
    Reason:                NotFound
    Status:                False
    Type:                  Ready
Events:
  Type    Reason        Age   From          Message
kubectl describe certificate quickstart-example-tls   ----    ------        ----  ----
↪          -------
  Normal  Generated     18s   cert-manager  Generated new private key
  Normal  OrderCreated  18s   cert-manager  Created Order resource "quickstart-
↪example-tls-889745041"
```

You can see the current state of the ACME Order by running `kubectl describe` on the Order resource that cert-manager has created for your Certificate:

```
$ kubectl describe order quickstart-example-tls-889745041
...
Events:
  Type    Reason      Age   From          Message
  ----    ------      ----  ----          -------
  Normal  Created     90s   cert-manager  Created Challenge resource "quickstart-
↪example-tls-889745041-0" for domain "example.your-domain.com"
```

Here, we can see that cert-manager has created 1 'Challenge' resource to fulfil the Order. You can dig into the state of the current ACME challenge by running `kubectl describe` on the automatically created Challenge resource:

```
$ kubectl describe challenge quickstart-example-tls-889745041-0
...

Status:
  Presented:   true
  Processing:  true
  Reason:      Waiting for http-01 challenge propagation
  State:       pending
Events:
```

(continues on next page)

(continued from previous page)

```
  Type    Reason     Age   From          Message
  ----    ------     ----  ----          -------
  Normal  Started    15s   cert-manager  Challenge scheduled for processing
  Normal  Presented  14s   cert-manager  Presented challenge using http-01 challenge␣
↪mechanism
```

From above, we can see that the challenge has been 'presented' and cert-manager is waiting for the challenge record to propagate to the ingress controller. You should keep an eye out for new events on the challenge resource, as a 'success' event should be printed after a minute or so (depending on how fast your ingress controller is at updating rules):

```
$ kubectl describe challenge quickstart-example-tls-889745041-0
...

Status:
  Presented:   false
  Processing:  false
  Reason:      Successfully authorized domain
  State:       valid
Events:
  Type    Reason           Age   From          Message
  ----    ------           ----  ----          -------
  Normal  Started          71s   cert-manager  Challenge scheduled for processing
  Normal  Presented        70s   cert-manager  Presented challenge using http-01␣
↪challenge mechanism
  Normal  DomainVerified  2s    cert-manager  Domain "example.your-domain.com"␣
↪verified with "http-01" validation
```

Note: If your challenges are not becoming 'valid' and remain in the 'pending' state (or enter into a 'failed' state), it is likely there is some kind of configuration error. Read the *Challenge resource reference docs* for more information on debugging failing challenges.

Once the challenge(s) have been completed, their corresponding challenge resources will be *deleted*, and the 'Order' will be updated to reflect the new state of the Order:

```
$ kubectl describe order quickstart-example-tls-889745041
...
Events:
  Type    Reason      Age   From          Message
  ----    ------      ----  ----          -------
  Normal  Created     90s   cert-manager  Created Challenge resource "quickstart-
↪example-tls-889745041-0" for domain "example.your-domain.com"
  Normal  OrderValid  16s   cert-manager  Order completed successfully
```

Finally, the 'Certificate' resource will be updated to reflect the state of the issuance process. If all is well, you should be able to 'describe' the Certificate and see something like the below:

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

Status:
  Conditions:
    Last Transition Time:  2019-01-09T13:57:52Z
    Message:               Certificate is up to date and has not expired
    Reason:                Ready
    Status:                True
```

(continues on next page)

(continued from previous page)

```
    Type:                  Ready
  Not After:               2019-04-09T12:57:50Z
Events:
  Type    Reason         Age                From          Message
  ----    ------         ----               ----          -------
  Normal  Generated      11m                cert-manager  Generated new private key
  Normal  OrderCreated   11m                cert-manager  Created Order resource
↪"quickstart-example-tls-889745041"
  Normal  OrderComplete  10m                cert-manager  Order "quickstart-example-
↪tls-889745041" completed successfully
```

2.1.2 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*.

```yaml
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
```

(continues on next page)

(continued from previous page)

```
            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
  ----    ------          ----     ----          -------
  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.

2.1.3 Issuing an ACME certificate using HTTP validation

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*.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: letsencrypt-staging
  namespace: default
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: {}
```

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.

2.1.4 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

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. Deploy cert-manager

cert-manager should be deployed using Helm, according to our official *Get 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!

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
```

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`:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  # Adjust the name here accordingly
  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 from step 3
    privateKeySecretRef:
      name: letsencrypt-private-key
    # Enable the HTTP-01 challenge provider
    solvers:
    - http01:
        ingress:
          class: nginx
```

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
...
Status:
  Acme:
    Uri:  https://acme-staging-v02.api.letsencrypt.org/acme/acct/7571319
  Conditions:
    Last Transition Time:  2019-01-30T14:52:03Z
    Message:               The ACME account was registered with the ACME server
    Reason:                ACMEAccountRegistered
    Status:                True
    Type:                  Ready
```

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 \
    jetstack/cert-manager \
    --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.

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 Securing Ingresses with Venafi

This guide walks you through how to secure a Kubernetes **'Ingress'_** resource using the Venafi Issuer type.

Whilst stepping through, you will learn how to:

- Create an EKS cluster using eksctl
- Install cert-manager into the EKS cluster
- Deploy **'nginx-ingress'_** to expose applications running in the cluster
- Configure a Venafi Cloud issuer
- Configure cert-manager to secure your application traffic

While this guide focuses on EKS as a Kubernetes provisioner and Venafi as a Certificate issuer, the steps here should be generally re-usable for other Issuer types.

2.2.1 Prerequisites

- An AWS account
- kubectl installed
- Access to a publicly registered DNS zone
- A Venafi Cloud account and API credentials

2.2.2 Create an EKS cluster

If you already have a running EKS cluster you can skip this step and move onto deploying cert-manager.

eksctl is a tool that makes it easier to deploy and manage an EKS cluster.

Installation instructions for various platforms can be found in the eksctl installation instructions.

Once installed, you can create a basic cluster by running:

```
eksctl create cluster
```

This process may take up to 20 minutes to complete. Complete instructions on using eksctl can be found in the eksctl usage section

Once your cluster has been created, you should verify that your cluster is running correctly by running the following command:

```
kubectl get pods --all-namespaces
NAME                       READY   STATUS    RESTARTS   AGE
aws-node-8xpkp             1/1     Running   0          115s
aws-node-tflxs             1/1     Running   0          118s
coredns-694d9447b-66vlp    1/1     Running   0          23s
coredns-694d9447b-w5bg8    1/1     Running   0          23s
kube-proxy-4dvpj           1/1     Running   0          115s
kube-proxy-tpvht           1/1     Running   0          118s
```

You should see output similar to the above, with all pods in a Running state.

2.2.3 Installing cert-manager

There are no special requirements to note when installing cert-manager on EKS, so the regular *Running on Kubernetes* guide can be used to install cert-manager.

Please walk through the installation guide and return to this step once you have validated cert-manager is deployed correctly.

2.2.4 Installing ingress-nginx

A Kubernetes ingress controller is designed to be the access point for HTTP and HTTPS traffic to the software running within your cluster. The **ingress-nginx_** controller does this by providing an HTTP proxy service supported by your cloud provider's load balancer (in this case, a Network Load Balancer (NLB).

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

To deploy ingress-nginx using an ELB to expose the service, run the following:

```
# Deploy the AWS specific pre-requisite manifest
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/
↪deploy/static/provider/aws/service-nlb.yaml

# Deploy the 'generic' ingress-nginx manifest
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/
↪deploy/static/mandatory.yaml
```

You may have to wait up to 5 minutes for all the required components in your cluster and AWS account to become ready.

You can run the following command to determine the address that Amazon has assigned to your NLB:

```
kubectl get service -n ingress-nginx
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP
↪                                      PORT(S)                      AGE
ingress-nginx   LoadBalancer   10.100.52.175   a8c2870a5a8a311e9a9a10a2e7af57d7-
↪6c2ec8ede48726ab.elb.eu-west-1.amazonaws.com   80:31649/TCP,443:30567/TCP   4m10s
```

The *EXTERNAL-IP* field may say `<pending>` for a while. This indicates the NLB is still being created. Retry the command until an *EXTERNAL-IP* has been provisioned.

Once the *EXTERNAL-IP* is available, you should run the following command to verify that traffic is being correctly routed to ingress-nginx:

```
curl http://a8c2870a5a8a311e9a9a10a2e7af57d7-6c2ec8ede48726ab.elb.eu-west-1.amazonaws.
↪com/

<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>openresty/1.15.8.1</center>
</body>
</html>
```

Whilst the above message would normally indicate an error (the page not being found), in this instance it indicates that traffic is being correctly routed to the ingress-nginx service.

Note: Although the AWS Application Load Balancer (ALB) is a modern load balancer offered by AWS that can can be provisioned from within EKS, at the time of writing, the alb-ingress-controller; is only capable of serving sites using certificates stored in AWS Certificate Manager (ACM). Version 1.15 of Kubernetes should address multiple bug fixes for this controller and allow for TLS termination support.

2.2.5 Configure your DNS records

Now that our NLB has been provisioned, we should point our application's DNS records at the NLBs address.

Go into your DNS provider's console and set a CNAME record pointing to your NLB.

For the purposes of demonstration, we will assume in this guide you have created the following DNS entry:

```
example.com CNAME a8c2870a5a8a311e9a9a10a2e7af57d7-6c2ec8ede48726ab.elb.eu-west-1.
↪amazonaws.com
```

As you progress through the rest of this tutorial, please replace `example.com` with your own registered domain.

2.2.6 Deploying a demo application

For the purposes of this demo, we provide an example deployment which is a simple "hello world" website.

First, create a new namespace that will contain your application:

```
kubectl create namespace demo
namespace/demo created
```

Save the following YAML into a file named `demo-deployment.yaml`:

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: hello-kubernetes
  namespace: demo
spec:
  type: ClusterIP
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: hello-kubernetes
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-kubernetes
  namespace: demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello-kubernetes
  template:
    metadata:
      labels:
        app: hello-kubernetes
    spec:
      containers:
      - name: hello-kubernetes
        image: paulbouwer/hello-kubernetes:1.5
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 8080
```

Then run:

```
kubectl apply -n demo -f demo-deployment.yaml
```

Note that the Service resource we deploy is of type ClusterIP and not LoadBalancer, as we will expose and secure traffic for this service using ingress-nginx that we deployed earlier.

You should be able to see two Pods and one Service in the `demo` namespace:

```
kubectl get po,svc -n demo
NAME                                     READY   STATUS    RESTARTS   AGE
hello-kubernetes-66d45d6dff-m2lnr   1/1     Running   0          7s
hello-kubernetes-66d45d6dff-qt2kb   1/1     Running   0          7s

NAME                         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/hello-kubernetes   ClusterIP   10.100.164.58   <none>        80/TCP     7s
```

Note that we have not yet exposed this application to be accessible over the internet. We will expose the demo

application to the internet in later steps.

2.2.7 Creating a Venafi Issuer resource

cert-manager supports both Venafi TPP and Venafi Cloud.

Please only follow one of the below sections according to where you want to retrieve your Certificates from.

Venafi TPP

Assuming you already have a Venafi TPP server set up properly, you can create a Venafi Issuer resource that can be used to issue certificates.

To do this, you need to make sure you have your TPP *username* and *password*.

In order for cert-manager to be able to authenticate with your Venafi TPP server and set up an Issuer resource, you'll need to create a Kubernetes Secret containing your username and password:

```
kubectl create secret generic \
     venafi-tpp-secret \
     --namespace=demo \
     --from-literal=username='YOUR_TPP_USERNAME_HERE' \
     --from-literal=password='YOUR_TPP_PASSWORD_HERE'
```

We must then create a Venafi Issuer resource, which represents a certificate authority within Kubernetes.

Save the following YAML into a file named `venafi-issuer.yaml`:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: venafi-issuer
  namespace: demo
spec:
  venafi:
    zone: "Default" # Set this to the Venafi policy zone you want to use
    tpp:
      url: https://venafi-tpp.example.com/vedsdk # Change this to the URL of your TPP␣
↪instance
      caBundle: <base64 encoded string of caBundle PEM file, or empty to use system␣
↪root CAs>
      credentialsRef:
        name: venafi-tpp-secret
```

Then run:

```
kubectl apply -n demo -f venafi-issuer.yaml
```

When you run the following command, you should see that the Status stanza of the output shows that the Issuer is Ready (i.e. has successfully validated itself with the Venafi TPP server).

```
kubectl describe issuer -n demo venafi-issuer

Status:
  Conditions:
    Last Transition Time:  2019-07-17T15:46:00Z
    Message:               Venafi issuer started
```

(continues on next page)

(continued from previous page)

```
    Reason:                Venafi issuer started
    Status:                True
    Type:                  Ready
Events:
  Type    Reason  Age   From          Message
  ----    ------  ----  ----          -------
  Normal  Ready   14s   cert-manager  Verified issuer with Venafi server
```

Venafi Cloud

You can sign up for a Venafi Cloud account by visiting the enroll page.

Once registered, you should fetch your API key by clicking your name in the top right of the control panel interface.

In order for cert-manager to be able to authenticate with your Venafi Cloud account and set up an Issuer resource, you'll need to create a Kubernetes Secret containing your API key:

```
kubectl create secret generic \
  venafi-cloud-secret \
  --namespace=demo \
  --from-literal=apikey=<API_KEY>
```

We must then create a Venafi Issuer resource, which represents a certificate authority within Kubernetes.

Save the following YAML into a file named `venafi-issuer.yaml`:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: venafi-issuer
  namespace: demo
spec:
  venafi:
    zone: "Default" # Set this to the Venafi policy zone you want to use
    cloud:
      url: "https://api.venafi.cloud/v1"
      apiTokenSecretRef:
        name: venafi-cloud-secret
        key: apikey
```

Then run:

```
kubectl apply -n demo -f venafi-issuer.yaml
```

When you run the following command, you should see that the Status stanza of the output shows that the Issuer is Ready (i.e. has successfully validated itself with the Venafi Cloud service).

```
kubectl describe issuer -n demo venafi-issuer

Status:
  Conditions:
    Last Transition Time:  2019-07-17T15:46:00Z
    Message:               Venafi issuer started
    Reason:                Venafi issuer started
    Status:                True
    Type:                  Ready
```

(continues on next page)

(continued from previous page)

```
Events:
  Type    Reason  Age   From          Message
  ----    ------  ----  ----          -------
  Normal  Ready   14s   cert-manager  Verified issuer with Venafi server
```

2.2.8 Request a Certificate

Now that the Issuer is configured and we have confirmed it has been set up correctly, we can begin requesting certificates which can be used by Kubernetes applications.

Full information on how to specify and request Certificate resources can be found in the *Issuing certificates* guide.

For now, we will create a basic x509 Certificate that is valid for our domain, `example.com`:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example-com-tls
  namespace: demo
spec:
  secretName: example-com-tls
  dnsNames:
  - example.com
  issuerRef:
    name: venafi-issuer
```

Save this YAML into a file named `example-com-tls.yaml` and run:

```
kubectl apply -n demo -f example-com-tls.yaml
```

As long as you've ensured that the zone of your Venafi Cloud account (in our example, we use the "Default" zone) has been configured with a CA or contains a custom certificate, cert-manager can now take steps to populate the `example-com-tls` Secret with a certificate. It does this by identifying itself with Venafi Cloud using the API key, then requesting a certificate to match the specifications of the Certificate resource that we've created.

You can run `kubectl describe` to check the progress of your Certificate:

```
kubectl describe certificate -n demo example-com-tls

...
Status:
  Conditions:
    Last Transition Time:  2019-07-17T17:43:01Z
    Message:               Certificate is up to date and has not expired
    Reason:                Ready
    Status:                True
    Type:                  Ready
  Not After:               2019-10-15T12:00:00Z
Events:
  Type    Reason       Age   From          Message
  ----    ------       ----  ----          -------
  Normal  Issuing      33s   cert-manager  Requesting new certificate...
  Normal  GenerateKey  33s   cert-manager  Generated new private key
  Normal  Validate     33s   cert-manager  Validated certificate request against␣
↪Venafi zone policy
  Normal  Requesting   33s   cert-manager  Requesting certificate from Venafi server..
↪.
```

(continues on next page)

(continued from previous page)

```
  Normal  Retrieve     15s   cert-manager  Retrieved certificate from Venafi server
  Normal  CertIssued   15s   cert-manager  Certificate issued successfully
```

Once the Certificate has been issued, you should see events similar to above.

You should then be able to see the certificate has been successfully stored in the Secret resource:

```
kubectl get secret -n demo example-com-tls

NAME              TYPE                DATA   AGE
example-com-tls   kubernetes.io/tls   3      2m47s

kubectl get secret example-com-tls -o 'go-template={{index .data "tls.crt"}}' | \
  base64 --decode | \
  openssl x509 -noout -text

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            0d:ce:bf:89:04:d4:41:83:f4:4c:32:66:64:fb:60:14
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=DigiCert Inc, CN=DigiCert Test SHA2 Intermediate CA-1
        Validity
            Not Before: Jul 17 00:00:00 2019 GMT
            Not After : Oct 15 12:00:00 2019 GMT
        Subject: C=US, ST=California, L=Palo Alto, O=Venafi Cloud, OU=SerialNumber,␣
↪CN=example.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:ad:2e:66:02:20:c9:b1:6a:00:63:70:4e:22:3c:
                    45:63:6e:e7:fd:4c:94:7d:75:50:22:a2:01:72:99:
                    9c:23:04:90:51:85:4d:47:32:e4:8b:ee:b1:ea:09:
                    1a:de:97:5d:31:05:a2:73:73:4f:06:a3:b2:59:ee:
                    bc:30:f7:26:85:3d:b3:56:e4:c2:97:34:b6:ac:6d:
                    65:7e:a2:4e:b4:ce:f2:0a:0a:4c:d7:32:d7:5a:18:
                    e8:69:c6:34:28:26:36:ef:c5:bc:ae:ba:ca:d2:46:
                    3f:d4:61:39:66:8f:19:cc:d6:d6:10:77:af:51:93:
                    1b:4d:f8:d1:10:19:ab:ac:b3:7b:0b:98:58:29:e6:
                    a9:ac:9f:7a:dc:63:0d:51:f5:bd:9f:f3:03:2e:b3:
                    2d:2f:00:87:f4:e1:cd:5a:32:c6:d8:fb:49:c4:e7:
                    da:3f:0f:8f:bb:66:94:28:5d:99:fe:7c:f0:17:1b:
                    fd:3e:ed:dd:36:bf:8e:62:60:0c:85:7f:76:74:4b:
                    37:d9:c2:e8:74:49:04:bf:f1:83:81:cc:4f:9b:f3:
                    40:97:d4:dc:b6:d3:2d:dc:73:18:93:48:a5:8f:6c:
                    57:7f:ec:62:c0:bc:c2:b0:e9:0a:51:2d:c4:b6:87:
                    68:96:87:f8:9a:86:3c:6a:f1:01:ca:57:c4:07:e7:
                    b0:51
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Authority Key Identifier:
                keyid:D6:4D:F9:39:60:6C:73:C3:22:F5:AD:30:0C:2F:A0:D5:CA:75:4A:2A

            X509v3 Subject Key Identifier:
                A3:B3:47:2C:41:5E:9C:B2:27:97:57:14:A4:2E:BA:8C:93:E7:01:65
```

(continues on next page)

(continued from previous page)

```
            X509v3 Subject Alternative Name:
                DNS:example.com
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 CRL Distribution Points:

                Full Name:
                  URI:http://crl3.digicert.com/DigiCertTestSHA2IntermediateCA1.crl

                Full Name:
                  URI:http://crl4.digicert.com/DigiCertTestSHA2IntermediateCA1.crl

            X509v3 Certificate Policies:
                Policy: 2.16.840.1.114412.1.1
                  CPS: https://www.digicert.com/CPS

            Authority Information Access:
                OCSP - URI:http://ocsp.digicert.com
                CA Issuers - URI:http://cacerts.test.digicert.com/
↪DigiCertTestSHA2IntermediateCA1.crt

            X509v3 Basic Constraints: critical
                CA:FALSE
    Signature Algorithm: sha256WithRSAEncryption
         ae:d4:9c:8a:66:19:9e:7d:12:b7:05:c2:b6:33:b3:9c:a5:40:
         47:ab:34:8d:1b:0f:51:96:de:e9:46:5a:e4:16:10:43:56:bf:
         fa:f8:64:f4:cb:53:39:5b:45:ca:7f:15:d9:59:25:21:23:c4:
         4d:dc:a7:f7:83:21:d2:3f:a8:0a:26:f4:ef:fa:1b:2b:7d:97:
         7e:28:f3:ca:cd:b2:c4:92:f3:92:27:7f:e0:f1:ac:d6:db:4c:
         10:8a:f8:6f:09:bb:b3:4f:19:06:aa:bb:74:1c:e0:51:42:f6:
         8c:7d:77:f7:80:a4:03:ab:a9:ae:ae:2b:89:17:af:2f:eb:f7:
         3d:61:7c:dd:e1:5d:d2:5a:c5:6a:f6:c8:92:4c:0a:b5:75:d1:
         dd:39:f2:a7:a2:10:8c:6d:bf:ca:08:ad:b9:a9:df:e3:59:8f:
         64:16:3c:7e:8a:6e:27:fc:49:d7:06:f0:bd:94:15:f2:fd:0f:
         94:8a:b8:73:67:73:53:22:df:9d:36:e9:34:f9:2a:68:00:59:
         78:6d:2d:8f:a0:0f:13:af:bd:b3:aa:8c:37:c4:22:cf:23:fb:
         56:bc:4e:55:ae:3a:0a:e6:3e:b1:1a:22:71:7b:08:b8:00:41:
         14:26:f6:9b:9b:72:3f:eb:dc:dd:1b:db:a8:20:fd:54:75:ae:
         25:7f:80:e6
```

In the next step, we'll configure your application to actually use this new Certificate resource.

2.2.9 Exposing and securing your application

Now that we have issued a Certificate, we can expose our application using a Kubernetes Ingress resource.

Create a file named `application-ingress.yaml` and save the following in it, replacing `example.com` with your own domain name:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend-ingress
  namespace: demo
```

(continues on next page)

(continued from previous page)

```
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  tls:
  - hosts:
    - example.com
    secretName: example-com-tls
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: hello-kubernetes
          servicePort: 80
```

You can then apply this resource with:

```
kubectl apply -n demo -f application-ingress.yaml
```

Once this has been created, you should be able to visit your application at the configured hostname, here `example.com`!

Navigate to the address in your web browser and you should see the certificate obtained via Venafi being used to secure application traffic.

CHAPTER 3

Tasks

This section contains guides on using specific features of cert-manager, such as configuring different Issuer types and any special settings that you may want to configure.

3.1 Setting up Issuers

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

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

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

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

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.

3.1.1 Supported issuer types

cert-manager supports a number of different issuer backends, each with their own different types of configuration.

Please follow one of the below linked guides to learn how to set up the issuer types you require:

- *CA* - issue certificates signed by a X509 signing keypair, stored in a Secret in the Kubernetes API server.
- *Self signed* - issue self signed certificates.
- *ACME* - issue certificates obtained by performing challenge validations against an ACME server such as Let's Encrypt.

- *Vault*- issue certificates from a Vault instance configured with the Vault PKI backend.
- *Venafi* - issue certificates from a Venafi Cloud or Trust Protection Platform instance.

3.1.2 Additional information

There are a few key things to know about Issuers, but for full information you can refer to the *Issuer reference docs*.

Difference between Issuers and 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 resource spec is identical, and you should set the `certificate.spec.issuerRef.kind` field to ClusterIssuer when creating your Certificate resources.

Setting up ACME Issuers

The ACME Issuer type represents a single Account registered with the ACME server.

When you create a new ACME Issuer, cert-manager will generate a private key which is used to identify you with the ACME server.

To set up a basic ACME issuer, you should create a new Issuer or ClusterIssuer resource.

You should read the guides linked at the bottom of this page to learn more about the ACME challenge validation mechanisms that cert-manager supports and how to configure the various DNS01 provider implementations.

Creating a basic ACME Issuer

The below example configures a ClusterIssuer named `letsencrypt-staging` that is configured to HTTP01 challenge solving with configuration suitable for ingress controllers such as **ingress-nginx_**.

You should copy and paste this example into a new file named `letsencrypt-staging.yaml` and update the `spec.acme.email` field to be your own email address.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
  acme:
    # You must replace this email address with your own.
    # Let's Encrypt will use this to contact you about expiring
    # certificates, and issues related to your account.
    email: user@example.com
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      # Secret resource used to store the account's private key.
      name: example-issuer-account-key
    # Add a single challenge solver, HTTP01 using nginx
    solvers:
```

(continues on next page)

(continued from previous page)

```
    - http01:
        ingress:
          class: nginx
```

You can then create this resource using `kubectl apply`:

```
kubectl apply -f letsencrypt-staging.yaml
```

To verify that the account has been registered successfully, you can run `kubectl describe` and check the 'Ready' condition:

```
kubectl describe clusterissuer letsencrypt-staging
...
Status:
  Acme:
    Uri:  https://acme-staging-v02.api.letsencrypt.org/acme/acct/7571319
  Conditions:
    Last Transition Time:  2019-01-30T14:52:03Z
    Message:               The ACME account was registered with the ACME server
    Reason:                ACMEAccountRegistered
    Status:                True
    Type:                  Ready
```

Any Certificate you create that references this Issuer resource will use the HTTP01 challenge solver you have configured above.

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.

Adding multiple solver types

You may want to use different types of challenge solver configuration for different ingress controllers, for example if you want to issue wildcard certificates using DNS01 alongside other certificates that are validated using HTTP01.

The `solvers` stanza has an optional `selector` field, that can be used to specify which Certificates, and further, what DNS names **on those certificates** should be used to solve challenges.

For example, to configure HTTP01 using nginx ingress as the default solver, along with a DNS01 solver that can be used for wildcard certificates:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
  acme:
    ...
    solvers:
    - http01:
        ingress:
          class: nginx
    - selector:
        matchLabels:
```

(continues on next page)

(continued from previous page)

```
          use-cloudflare-solver: "true"
      dns01:
        cloudflare:
          email: user@example.com
          apiKeySecretRef:
            name: cloudflare-apikey-secret
            key: apikey
```

In order to utilise the configured cloudflare DNS01 solver, you must add the `use-cloudflare-solver: "true"` label to your Certificate resources.

Using multiple solvers for a single certificate

The solver's `selector` stanza has an additional field `dnsNames` that further refines the set of domains that the solver configuration applies to.

If any `dnsNames` are specified, then that challenge solver will be used if the domain being validated is named in that list.

For example:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
  acme:
    ...
    solvers:
    - http01:
        ingress:
          class: nginx
    - selector:
        dnsNames:
        - '*.example.com'
      dns01:
        cloudflare:
          email: user@example.com
          apiKeySecretRef:
            name: cloudflare-apikey-secret
            key: apikey
```

In this instance, a Certificate that specified both `*.example.com` and `example.com` would use the HTTP01 challenge solver for `example.com` and the DNS01 challenge solver for `*.example.com`.

It is possible to specify both `matchLabels` AND `dnsNames` on an ACME solver selector.

Configuring HTTP01 Ingress Provider

This page contains details on the different options available on the `Issuer` resource's HTTP01 challenge solver configuration.

For more information on configuring ACME issuers and their API format, read the *Setting up ACME Issuers* documentation.

How HTTP01 validations work

You can read about how the HTTP01 challenge type works on the Let's Encrypt challenge types page.

Options

The HTTP01 Issuer supports a number of additional options. For full details on the range of options available, read the reference documentation.

ingressClass

If the `ingressClass` field is specified, cert-manager will create new Ingress resources in order to route traffic to the 'acmesolver' pods, which are responsible for responding to ACME challenge validation requests.

If this field is not specified, and `ingressName` is also not specified, cert-manager will default to create **new** ingress resources but will **not** set the ingress class on these resources, meaning **all** ingress controllers installed in your cluster will server traffic for the challenge solver, potentially occurring additional cost.

ingressName

If the 'ingressName' field is specified, cert-manager will edit the named ingress resource in order to solve HTTP01 challenges.

This is useful for compatibility with ingress controllers such as **ingress-gce_**, which utilise a unique IP address for each Ingress resource created.

This mode should be avoided when using ingress controllers that expose a single IP for all ingress resources, as it can create compatibility problems with certain ingress-controller specific annotations.

servicePort

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.

podTemplate

You may wish to change or add to the labels and annotations of solver pods. These can be configured under the `metadata` field under `podTemplate`.

Similarly, you can set the nodeSelector, tolerations and affinity of solver pods by configuring under the `spec` field of the `podTemplate`. No other spec fields can be edited.

An example of how you could configure the template is as so:

```yaml
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: ...
spec:
  acme:
    server: ...
    privateKeySecretRef:
      name: ...
    solvers:
    - http01:
        ingress:
          podTemplate:
            metadata:
              labels:
                foo: "bar"
                env: "prod"
            spec:
              nodeSelector:
                bar: baz
```

The added labels and annotations will merge on top of the cert-manager defaults, overriding entries with the same key.

No other fields can be edited.

Configuring DNS01 Challenge Providers

This page contains details on the different options available on the `Issuer` resource's DNS01 challenge solver configuration.

For more information on configuring ACME issuers and their API format, read the *Setting up ACME Issuers* documentation.

DNS01 provider configuration must be specified on the Issuer resource, similar to the examples in the setting up documentation:

You can read about how the DNS01 challenge type works on the Let's Encrypt challenge types page.

```yaml
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
    solvers:
    - dns01:
        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).

For more information on utilising multiple solver types on a single Issuer, read the **multiple-solver-types_** section.

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 exposes a flag that allows you alter this behaviour:

Example usage:

```
--dns01-recursive-nameservers "8.8.8.8:53,1.1.1.1:53"
```

Delegated Domains for DNS01

By default, cert-manager will not follow CNAME records pointing to subdomains.

If granting cert-manager access to the root DNS zone is not desired, then the _acme-challenge.example.com subdomain can instead be delegated to some other, less privileged domain. Once a CNAME record has been configured to point at the desired domain, and the DNS configuration/credentials for the zone that *should be updated* have been provided, all that is left to be done is adding an additional field into the relevant *dns01* solver:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  ...
spec:
  acme:
    ...
    solvers:
    - dns01:
        # Valid values are None and Follow
        cnameStrategy: Follow
        clouddns:
          ...
```

cert-manager will then follow CNAME records recursively in order to determine which DNS zone to update during DNS01 challenges.

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

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: example-issuer
spec:
```

(continues on next page)

(continued from previous page)

```
  acme:
    ...
    solvers:
    - dns01:
        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"]
   ```

(continues on next page)

(continued from previous page)

```
  },
  "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

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: example-issuer
spec:
  acme:
    ...
    solvers:
    - dns01:
        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 you can use the following script (requires `azure-cli` and `jq`):

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

DNS_SP=$(az ad sp create-for-rbac --name $AZURE_CERT_MANAGER_SP_NAME)
AZURE_CERT_MANAGER_SP_APP_ID=$(echo $DNS_SP | jq -r '.appId')
AZURE_CERT_MANAGER_SP_PASSWORD=$(echo $DNS_SP | jq -r '.password')

# 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

# 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 "Principal: $AZURE_CERT_MANAGER_SP_APP_ID"
echo "Password: $AZURE_CERT_MANAGER_SP_PASSWORD"
```

You can configure the issuer like so:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: example-issuer
spec:
  acme:
    ...
    solvers:
    - dns01:
        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:
```

(continues on next page)

(continued from previous page)

```
            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
          # Azure Cloud Environment, default to AzurePublicCloud
          environment: AZURE_ENVIRONMENT
```

Cloudflare

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: example-issuer
spec:
  acme:
    ...
    solvers:
    - dns01:
        cloudflare:
          email: my-cloudflare-acc@example.com
          apiKeySecretRef:
            name: cloudflare-api-key-secret
            key: api-key
```

Google CloudDNS

This guide explains how to set up an Issuer, or ClusterIssuer, to use Google CloudDNS to solve DNS01 ACME challenges. It's advised you read the *DNS01 Challenge Provider* page first for a more general understanding of how cert-manager handles DNS01 challenges.

Note: This guide assumes that your cluster is hosted on Google Cloud Platform (GCP) and that you already have a domain set up with CloudDNS.

Set up a Service Account

Cert-manager needs to be able to add records to CloudDNS in order to solve the DNS01 challenge. To enable this, a GCP service account must be created with the `dns.admin` role.

Note: For this guide the `gcloud` command will be used to set up the service account. Ensure that `gcloud` is in using the correct project and zone before entering the commands. These steps could also be completed using the Cloud Console.

Create a Service Account Secret

To access this service account cert-manager uses a key stored in a Kubernetes Secret. First, create a key for the service account and download it as JSON file, then create a Secret from this file.

If you did not create the service "dns01-solver" account before, you need to create it first:

```
gcloud iam service-accounts create dns01-solver
```

```
# Replace use of project-id with the id of your project
gcloud iam service-accounts keys create key.json \
 --iam-account dns01-solver@$PROJECT_ID.iam.gserviceaccount.com
kubectl create secret generic clouddns-dns01-solver-svc-acct \
 --from-file=key.json
```

Note: Keep the key file safe and do not share it, as it could be used to gain access to your cloud resources. The key file can be deleted once it has been used to generate the Secret.

Note: If you have already added the secret but get an error: *... due to error processing: error getting clouddns service account: secret "XXX" not found*, the secret may be in the wrong namespace. If you're configuring a *ClusterIssuer*, try moving the secret to the same namespace as cert-manager. If you're configuring an *Issuer*, the secret should be stored in the same namespace as the *Issuer* resource.

Create an Issuer That Uses CloudDNS

Next, create an Issuer (or ClusterIssuer) with a `clouddns` provider. An example Issuer manifest can be seen below with annotations.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: example-issuer
spec:
  acme:
    ...
    solvers:
    - dns01:
        clouddns:
          # The ID of the GCP project
          project: $PROJECT_ID
          # This is the secret used to access the service account
          serviceAccountSecretRef:
            name: clouddns-dns01-solver-svc-acct
            key: key.json
```

For more information about Issuers and ClusterIssuers, see *Setting Up Issuers*.

Once an Issuer (or ClusterIssuer) has been created successfully a Certificate can then be added to verify that everything works.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
```

(continues on next page)

(continued from previous page)

```
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  issuerRef:
    # The issuer created previously
    name: example-issuer
  commonName: example.com
  dnsNames:
  - example.com
  - www.example.com
```

For more details about Certificates, see *Issuing Certificates*.

Amazon Route53

This guide explains how to set up an Issuer, or ClusterIssuer, to use Amazon Route53 to solve DNS01 ACME challenges. It's advised you read the *DNS01 Challenge Provider* page first for a more general understanding of how cert-manager handles DNS01 challenges.

Note: This guide assumes that your cluster is hosted on Amazon Web Services (AWS) and that you already have a hosted zone in Route53.

Set up a IAM Role

Cert-manager needs to be able to add records to Route53 in order to solve the DNS01 challenge. To enable this, create a IAM policy with the following permissions:

```
{
    "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": "*"
        }
    ]
}
```

Note: The `route53:ListHostedZonesByName` statement can be removed if you specify the (optional)

`hostedZoneID`. You can further tighten the policy by limiting the hosted zone that cert-manager has access to (e.g. `arn:aws:route53:::hostedzone/DIKER8JEXAMPLE`).

Credentials

You have two options for the set up: Either create a user or a role and attach that policy from above. Using a role is considered best practice because you do not have to store permanent credentials in a secret.

Cert-manager supports two ways of specifying credentials:

- explicit by providing a `accessKeyID` and `secretAccessKey`
- or implicit (using metadata service or env vars or credentials file)

Cert-manager also supports specifying a `role` to enable cross-account access and/or to limit the access for the cert-manager. Integration with kiam and kube2iam should work out of the box.

Cross account access

Example: Account A manages a Route53 DNS Zone. Now you want account X to be able to manage records in that zone.

First, create a role with the policy above (let's call the role `dns-manager`) and attach a trust relationship like the one below. Make sure role `cert-manager` in account X exists:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::XXXXXXXXXXX:role/cert-manager"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

This allows the role `cert-manager` in account X to manage the Route53 DNS Zone in account A. For more information visit the official documentation.

Creating a Issuer (or ClusterIssuer)

Here is an example configuration for a ClusterIssuer:

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

(continues on next page)

(continued from previous page)

```
    # example: cross-account zone management for example.com
    # this solver uses ambient credentials (i.e. inferred from the environment or EC2␣
↪Metadata Service)
    # to assume a role in a different account
    - selector:
        dnsZones:
          - "example.com"
      dns01:
        route53:
          region: us-east-1
          hostedZoneID: DIKER8JEXAMPLE # optional, see bpolicy above
          role: arn:aws:iam::XXXXXXXXXXXX:role/dns-manager

    # this solver handles foobar.cloud challenges
    # and uses explicit credentials
    - selector:
        dnsZones:
          - "foobar.cloud"
      dns01:
        route53:
          region: eu-central-1
          accessKeyID: AKIAIOSFODNN7EXAMPLE
          secretAccessKeySecretRef:
            name: prod-route53-credentials-secret
            key: secret-access-key
          # you can also assume a role with these credentials
          role: arn:aws:iam::XXXXXXXXXXXX:role/dns-manager
```

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

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: example-issuer
spec:
  acme:
    ...
    solvers:
    - dns01:
        digitalocean:
          tokenSecretRef:
            name: digitalocean-dns
            key: access-token
```

RFC-2136

The goal of this document is to provide a configuration overview of the various facilities required to deploy cert-manager against a RFC-2136 compliant DNS server such as BIND `named`. This capability is also commonly known

as “dynamic DNS”.

Unlike the peer of other cert-manager DNS integrations, `named` is a bit of a “Swiss Army Knife” of domain name servers. Over the years, it has been highly optimized to provide maximal vertical scalability for a single node, as well as horizontal scalability with service provider interfaces. This flexibility makes it impossible to go into every possible `named` deployment that a user may run in to though. Instead, this document will try to make sure your server is ready to accept requests from cert-manager using command line tools, then get on to the making the two work together.

Transaction Signatures TSIG

Dynamic DNS updates are essentially server queries which otherwise might return resource records (RRs). Since DNS servers are commonly exposed to the public internet, being able to push an unauthenticated update to any server that responds to queries would be immediately untenable.

In the eyes of the `named` architects, the generic solution to this problem space was twofold. The first is to require manual enablement of updates at a zone level, such as `example.com`. In a naive network, there is no requirement that zone updates have any security to them, and clients can be configured such that they can provide updates without any authentication. An example of where this is useful is for machines booting using DHCP, in this case the machines know about themselves and the DNS server can be configured to accept updates when they come from the address being configured.

This clearly has limitations in situations such as cert-manager and the DNS-01 challenge. In this environment, a TXT RR must be created after coordination with the ACME server. After negotiating with the ACME server, a the TXT RR that is published on the domain validates that the domain is legitimately engaged with the process of creating a certificate for it. In the bigger picture of DNS, this means that an arbitrary actor (cert-manager, in this case) must be able to add one of these KV mappings to the domain and delete it after the certificate has been issued. `cert-manager` does not have a convenient physical characteristic such as a DHCP allocation to validate it’s requests.

For cases like this, we need to be able to sign a request that is being sent to the DNS server. We do that through TSIGs, or Transaction SIGnatures.

Configuration Step 1 - Set up your DNS server for secure dynamic updates

There are many excellent tutorials on the net that walk through preparing a basic `named` server for dynamic updates:

- https://www.cyberciti.biz/faq/unix-linux-bind-named-configuring-tsig/
- https://tomthorp.me/blog/using-tsig-enable-secure-zone-transfers-between-bind-9x-servers

More complex `named` deployments will not use text files, but rather may use LDAP or SQL for a database for resource records. An additional wrinkle is metadata configuration, such as for zone metadata like enabling dynamic updates or access control lists (ACLs) for a zone. There are too many configurations to go into here, but you should be able to find the documentation to do so.

Whatever your deployment is, the goal at this stage has nothing to do with cert-manager and everything to do with a tool called `nsupdate` generating updates signed with TSIG. Once this is out of the way, you can attack the cert-manager configuration with far greater confidence.

Using `nsupdate`

Most paths to configuring BIND `named` will go through using `dnssec-keygen`. This command-line tool generates a named private key that is used for signing TSIG requests. When a request is signed, both the signature and the name of the private key are attached to the request in an unencrypted form. In this manner, when the request is received, the

name of the private key can be used to by the recipient to find the private key itself, build a new signature with it, and compare the two for acceptance.

Since there are dozens of ways to have your `named` server misconfigured, we'll use `nsupdate` to test that the server behaves as expected before we get there. https://debian-administration.org/article/591/Using_the_dynamic_DNS_editor_nsupdate is a solid breakdown of how to use the tool.

To get started, we'll simply run `nsupdate -k <keyID>` where keyID is the value returned from `dnssec-keygen`. This will read the key from disk and provide a command prompt to issue commands. In general, we want to write a simple TXT RR and make sure we can delete it.

```
$ nsupdate -k <keyID>
> update add www1.example.com txt testing
> send
> ... test here with ``nslookup``
> update delete www1.example.com txt
> send
> ... test here with ``nslookup``
```

Any failures to write, read or delete the record will mean that cert-manager will not be able to do so either, no matter how well it is configured.

Configuration Step 2 - Set up cert-manager

Now we get to the fun stuff, seeing everything work. Remember that we need to set up the ACME DNS-01 issuer and challenge mechanism as well as the `rfc2136` provider. Since the documentation covers the other parts sufficiently, let's focus on the provider here.

For example:

```
rfc2136:
  nameserver: 1.2.3.4:53
  tsigKeyName: example-com-secret
  tsigAlgorithm: HMACSHA512
  tsigSecretSecretRef:
    name: tsig-secret
    key: tsig-secret-key
```

For this example configuration, we'll need the following two commands. The first, on your `named` server generates the key. Note how `example-com-secret` is both in the `tsigKeyName` above and the `dnssec-keygen` command that follows.

```
dnssec-keygen -r /dev/urandom -a HMAC-SHA512 -b 512 -n HOST example-com-secret
```

Also note how the `tsigAlgorithm` is provided in both the configuration and the keygen command. They are listed at https://github.com/miekg/dns/blob/v1.0.12/tsig.go#L18-L23.

The second bit of configuration you need on the kubernetes side is to create a secret. Pulling the secret key string from the `<key>.private` file generated above, use the secret in the placeholder below:

```
kubectl -n cert-manager create secret generic tsig-secret --from-literal=tsig-secret-
↪key=<somesecret>
```

Note how the `tsig-secret` and `tsig-secret-key` match the configuration in the `tsigSecretSecretRef` above.

Rate Limits

The `rfc2136` provider waits until *all* nameservers to in your domain's SOA RR respond with the same result before it contacts Let's Encrypt to complete the challenge process. This is because the challenge server contacts a non-authoritative DNS server that does a recursive query (a query for records it does not maintain locally). If the servers in the SOA do not contain the correct values, it's likely that the non-authoritative server will have bad information as well, causing the request to go against rate limits and eventually locking the process out.

This process is in place to protect users from server misconfigurations creating a more subtle lockout that persists after the server configuration has been repaired.

As documented elsewhere, it is prudent to fully debug configurations using the ACME staging servers before using the production servers. The staging servers have less aggressive rate limits, but the certificates they issue are not signed with a root certificate trusted by browsers.

What's next?

This configuration so far will actually do nothing. You still have to request a certificate as in *Issuing Certificates*. Once a certficate is requested, the provider will begin processing the request.

Troubleshooting

- Be sure that you have fully tested the DNS server updates using `nsupdate` first. Ideally, this is done from a pod in the same namespace as the `rfc2136` provider to ensure there are no firewall issues.
- The logs for the `cert-manager` pod are your friend. Additional logs can be generated by adding the `--v=5` argument to the container launch.
- The TSIG key is encoded with `base64`, but the Kubernetes API server also expects that key literals will be decoded before they are stored. In some cases, a key must be double-encoded. (If you've tested using `nsupdate`, it's pretty easy to spot when you are running into this.)
- Pay attention to the refresh time of the zone you are working with. For zones with low traffic, it will not make a significant difference to reduce the refresh time down to about five minutes while getting initial certificates. Once the process is working, the beauty of `cert-manager` is it doesn't matter if a renewal takes hours due to refresh times, it's all automated!
- Compared to the other providers that often use REST APIs to modify DNS RRs, this provider can take a little longer. You can `watch kubectl certificate yourcert` to get a display of what's going on. It's not uncommon for the process to take five minutes in total.

Setting up CA Issuers

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.

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. 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
```

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!

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`).

The example above explicitly sets the `commonName` field to `example.com`. cert-manager automatically adds the `commonName` field as a DNS SAN if it is not already contained in the `dnsNames` field.

If we had **not** specified the `commonName` field, then the **first** DNS SAN that is specified (under `dnsNames`) would be used as the certificate's common name.

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.

Setting up self signing Issuers

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*.

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
  commonName: "my-selfsigned-root-ca"
  isCA: true
  issuerRef:
    name: selfsigning-issuer
    kind: ClusterIssuer
```

Setting up Vault Issuers

Installing Vault

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

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.

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:
          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 has 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*.

Vault Authentication with a Token

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

For testing purposes a root token is generated at Vault installation time. **WARNING: Root tokens do not expire, so should only be used for testing purposes.**

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*.

Setting up Venafi Issuers

The Venafi Issuer types allows you to obtain certificates from Venafi Cloud and Venafi Trust Protection Platform instances.

Register your account at https://ui.venafi.cloud/enroll and get an API key from your dashboard.

You can have multiple different Venafi Issuer types installed within the same cluster, including mixtures of Cloud and TPP issuer types. This allows you to be flexible with the types of Venafi account you use.

Automated certificate renewal and management are provided for Certificates using the Venafi issuer.

Note: The Venafi Issuer has been recently added, and the exact structure of the Issuer resource is subject to change. Such changes will be clearly documented, and migration steps will be provided.

Creating an Issuer resource

A single Venafi Issuer represents a single 'zone' within the Venafi API, therefore you must create an Issuer resource for each Venafi Zone you want to obtain certificates from.

You can configure your Issuer resource to either issue certificates only within a single namespace, or cluster-wide (using a ClusterIssuer resource). For more information on the distinction between Issuer and ClusterIssuer resources, read the *Difference between Issuers and ClusterIssuers* section.

Creating a Venafi Cloud Issuer

In order to set up a Venafi Cloud Issuer, you must first create a Kubernetes Secret resource containing your Venafi Cloud API credentials:

```
kubectl create secret generic \
     cloud-secret \
     --namespace='NAMESPACE OF YOUR ISSUER RESOURCE' \
     --from-literal=apikey='YOUR_CLOUD_API_KEY_HERE'
```

Note: If you are configuring your Issuer as a ClusterIssuer resource in order to issue Certificates across your whole cluster, you must set the `--namespace` parameter to `cert-manager`, which is the default 'cluster resource namespace'.

This API key will be used by cert-manager to interact with the Venafi Cloud service on your behalf.

Once the API key Secret has been created, you can create your Issuer or ClusterIssuer resource. If you are creating a ClusterIssuer resource, you must change the `kind` field to `ClusterIssuer` and remove the `metadata.namespace` field.

Save the below content after making your amendments to a file named `venafi-cloud-issuer.yaml`:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: cloud-venafi-issuer
  namespace: <NAMESPACE YOU WANT TO ISSUE CERTIFICATES IN>
spec:
  venafi:
    zone: "DevOps" # Set this to the Venafi policy zone you want to use
    cloud:
      apiTokenSecretRef:
        name: cloud-secret
        key: apikey
```

You can then create the Issuer using `kubectl create -f`:

```
kubectl create -f venafi-cloud-issuer.yaml
```

Verify the Issuer has been initialised correctly using `kubectl describe`:

```
kubectl describe issuer cloud-venafi-issuer --namespace='NAMESPACE OF YOUR ISSUER␣
↪RESOURCE'

(TODO) include sample output
```

You are now ready to issue certificates using the newly provisioned Venafi Issuer.

Read the *Issuing Certificates* document for more information on how to create Certificate resources.

Creating a Venafi Trust Protection Platform Issuer

The Venafi Trust Protection integration allows you to obtain certificates from a properly configured Venafi TPP instance.

The setup is similar to the Venafi Cloud configuration above, however some of the connection parameters are slightly different.

Note: You **must** allow "User Provided CSRs" as part of your TPP policy, as this is the only type supported by cert-manager at this time.

In order to set up a Venafi Trust Protection Platform Issuer, you must first create a Kubernetes Secret resource containing your Venafi TPP API credentials:

```
kubectl create secret generic \
     tpp-secret \
     --namespace=<NAMESPACE OF YOUR ISSUER RESOURCE> \
     --from-literal=username='YOUR_TPP_USERNAME_HERE' \
     --from-literal=password='YOUR_TPP_PASSWORD_HERE'
```

Note: If you are configuring your Issuer as a ClusterIssuer resource in order to issue Certificates across your whole cluster, you must set the `--namespace` parameter to `cert-manager`, which is the default 'cluster resource namespace'.

These credentials will be used by cert-manager to interact with your Venafi TPP instance. Username attribute must be adhere to the <identity provider>:<username> format. For example: `local:admin`.

Once the Secret containing credentials has been created, you can create your Issuer or ClusterIssuer resource. If you are creating a ClusterIssuer resource, you must change the `kind` field to `ClusterIssuer` and remove the `metadata.namespace` field.

Save the below content after making your amendments to a file named `venafi-tpp-issuer.yaml`:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: tpp-venafi-issuer
  namespace: <NAMESPACE YOU WANT TO ISSUE CERTIFICATES IN>
spec:
  venafi:
    zone: devops\cert-manager # Set this to the Venafi policy zone you want to use
    tpp:
      url: https://tpp.venafi.example/vedsdk # Change this to the URL of your TPP␣
↪instance
      caBundle: <base64 encoded string of caBundle PEM file, or empty to use system␣
↪root CAs>
      credentialsRef:
        name: tpp-secret
```

You can then create the Issuer using `kubectl create -f`:

```
kubectl create -f venafi-tpp-issuer.yaml
```

Verify the Issuer has been initialised correctly using `kubectl describe`:

```
kubectl describe issuer tpp-venafi-issuer --namespace='NAMESPACE OF YOUR ISSUER␣
↪RESOURCE'

(TODO) include sample output
```

You are now ready to issue certificates using the newly provisioned Venafi Issuer.

Read the *Issuing Certificates* document for more information on how to create Certificate resources.

3.2 Issuing Certificates

The Certificate resource type is used to request certificates from different Issuers.

In order to issue any certificates, you'll need to configure an Issuer resource first.

If you have not configured any issuers yet, you should read the *Setting up Issuers* guide.

3.2.1 Creating Certificate resources

A Certificate resource specifies fields that are used to generated certificate signing requests which are then fulfilled by the issuer type you have referenced.

Certificates specify which issuer they want to obtain the certificate from by specifying the `certificate.spec.issuerRef` field.

A basic Certificate resource, for the `example.com` and `www.example.com` DNS names that is valid for 90d and renews 15d before expiry is below:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  duration: 2160h # 90d
  renewBefore: 360h # 15d
  commonName: example.com
  dnsNames:
  - example.com
  - www.example.com
  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
```

The signed certificate will be stored in a Secret resource named `example-com-tls` once the issuer has successfully issued the requested certificate.

The Certificate will be issued using the issuer named `ca-issuer` in the `default` namespace (the same namespace as the Certificate resource).

Note: If you want to create an Issuer that can be referenced by Certificate resources in **all** namespaces, you should create a *ClusterIssuer* resource and set the `certificate.spec.issuerRef.kind` field to `ClusterIssuer`.

Note: The `renewBefore` and `duration` fields must be specified using Golang's `time.Time` string format, which does not allow the `d` (days) suffix. You must specify these values using `s`, `m` and `h` suffixes instead. Failing to

do so without installing the *webhook* component can prevent cert-manager from functioning correctly (#1269).

Note: Take care when setting the `renewBefore` field to be very close to the `duration` as this can lead to a renewal loop, where the Certificate is always in the renewal period. Some Issuers set the `notBefore` field on their issued X.509 certificate before the issue time to fix clock-skew issues, leading to the working duration of a certificate to be less than the full duration of the certificate. For example, Let's Encrypt sets it to be one hour before issue time, so the actual *working duration* of the certificate is 89 days, 23 hours (the *full duration* remains 90 days).

A full list of the fields supported on the Certificate resource can be found in the API reference documentation.

3.2.2 Temporary certificates whilst issuing

With some Issuer types, certificates can take a few minutes to be issued.

A temporary untrusted certificate will be issued whilst this process takes places if another certificate does not already exist in the target Secret resource.

This helps to improve compatibility with certain ingress controllers (e.g. ingress-gce) which require a TLS certificate to be present at all times in order to function.

After the real, valid certificate has been obtained, cert-manager will replace the temporary self signed certificate with the valid one, **but will retain the same private key**.

You can disable issuing temporary certificate by setting feature gate flag `--feature-gates=IssueTemporaryCertificate=false`

Automatically creating Certificates for Ingress resources

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.

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:
```

(continues on next page)

(continued from previous page)

```
      - 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.
```

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*.

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.
- `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-challenge-type` - (**DEPRECATED**) 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` - (**DEPRECATED**) 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` - (**DEPRECATED**) 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.

- `certmanager.k8s.io/acme-http01-edit-in-place: "true"` - (**DEPRECATED**) 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".

3.3 Backing up and restoring

If you need to uninstall cert-manager, or transfer your installation to a new cluster, you can backup all of cert-manager's configuration in order to later re-install.

3.3.1 Backing up

To backup all of your cert-manager configuration resources, run:

```
kubectl get -o yaml \
   --all-namespaces \
   issuer,clusterissuer,certificates,orders,challenges,certificaterequests > cert-
↪manager-backup.yaml
```

If you are transferring data to a new cluster, you may also need to copy across additional Secret resources that are referenced by your configured Issuers, such as:

CA Issuers

- The root CA Secret referenced by `issuer.spec.ca.secretName`

Vault Issuers

- The token authentication Secret referenced by `issuer.spec.vault.auth.tokenSecretRef`
- The approle configuration Secret referenced by `issuer.spec.vault.auth.appRole.secretRef`

ACME Issuers

- The ACME account private key Secret referenced by `issuer.acme.privateKeySecretRef`
- Any Secrets referenced by DNS providers configured under the `issuer.acme.dns01.providers` and `issuer.acme.solvers.dns01` fields.

3.3.2 Restoring

In order to restore your configuration, you can simply `kubectl apply` the files created above after installing cert-manager.

```
kubectl apply -f cert-manager-backup.yaml
```

If you have migrated from an old cluster, you will need to make sure to run a similar `kubectl apply` command to restore your Secret resources too.

3.4 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.

Note: Before performing upgrades of cert-manager, it is advised to take a backup of all your cert-manager resources just in case an issue occurs whilst upgrading. You can read how to backup and restore cert-manager in the *Backing up and restoring* guide.

3.4.1 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 upgrading 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`) and replacing `<version>` with the version number you want to install:

```
# Install the cert-manager CustomResourceDefinition resources before
# upgrading the Helm chart
kubectl apply \
     -f https://raw.githubusercontent.com/jetstack/cert-manager/<version>/deploy/
↪manifests/00-crds.yaml

# Add the Jetstack Helm repository if you haven't already
helm repo add jetstack https://charts.jetstack.io

# Ensure the local Helm chart repository cache is up to date
helm repo update

# If you are upgrading from v0.5 or below, you should manually add this
# label to your cert-manager namespace to ensure the `webhook component`_
# can provision correctly.
kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true

helm upgrade --version <version> <release_name> jetstack/cert-manager
```

This will upgrade you to the latest version of cert-manager, as listed in the **'Jetstack Helm chart repository'_**.

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

3.4.2 Upgrading using static manifests

If you installed cert-manager using the static deployment manifests published on each release, 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>` with the version number you want to install:

```
# If you are upgrading from v0.5 or below, you should manually add this
# label to your cert-manager namespace to ensure the `webhook component`_
# can provision correctly.
kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true

kubectl apply \
     -f https://github.com/jetstack/cert-manager/releases/download/<version>/cert-
↪manager.yaml
```

Note: If you are running kubectl v1.12 or below, you will need to add the `--validate=false` flag to your `kubectl apply` command above else you will receive a validation error relating to the `caBundle` field of the `ValidatingWebhookConfiguration` resource. This issue is resolved in Kubernetes 1.13 onwards. More details can be found in kubernetes/kubernetes#69590.

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.

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.

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>
```

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'
```

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
```

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"
```

Upgrading from v0.3 to v0.4

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

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.

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.

Upgrading from v0.5 to v0.6

> **Warning:** If you are upgrading from a release older than v0.5, please read the *Upgrading from older versions using Helm* note at the bottom of this document!

The upgrade process from v0.5 to v0.6 should be fairly seamless for most users. As part of the new release, we have changed how we ship the CustomResourceDefinition resources that cert-manager needs in order to operate (as well as introducing two **new** CRD types).

Depending on the way you have installed cert-manager in the past, your upgrade process will slightly vary:

Upgrading with the Helm chart

If you have previously deployed cert-manager v0.5 using the Helm installation method, you will now need to perform one extra step before upgrading.

Due to issues with the way Helm handles CRD resources in Helm charts, we have now moved the installation of these resources into a separate YAML manifest that must be installed with `kubectl apply` before upgrading the chart.

You can follow the *regular upgrade guide* as usual in order to upgrade from v0.5 to v0.6.

Upgrading with static manifests

The static manifests have moved into the `deploy/manifests` directory for this release.

We now also no longer ship different manifests for different configurations, in favour of a single `cert-manager.yaml` file which should work for all Kubernetes clusters from Kubernetes v1.9 onwards.

You can follow the *regular upgrade guide* as usual in order to upgrade from v0.5 to v0.6.

Upgrading from older versions using Helm

If you are upgrading from a version **older than v0.5** and **have installed with Helm**, you will need to perform a fresh installation of cert-manager due to issues with the Helm upgrade process. This will involve the **removal of all cert-manager custom resources**. This **will not** delete the Secret resources being used by your apps.

Before upgrading you will need to:

1. Read and follow the *backup guide* to create a backup of your configuration.
2. Delete the existing cert-manager Helm release (replacing 'cert-manager' with the name of your Helm release):

```
# Uninstall the Helm chart
$ helm delete --purge cert-manager

# Ensure the cert-manager CustomResourceDefinition resources do not exist:
$ kubectl delete crd \
    certificates.certmanager.k8s.io \
    issuers.certmanager.k8s.io \
    clusterissuers.certmanager.k8s.io
```

3. Perform a fresh install (as per the *installation guide*):

```
# Install the cert-manager CRDs
$ kubectl apply \
    -f https://raw.githubusercontent.com/jetstack/cert-manager/release-0.6/deploy/
↪manifests/00-crds.yaml

# Update helm repository cache
$ helm repo update

# Install cert-manager
$ helm install \
    --name cert-manager \
    --namespace cert-manager \
    --version v0.6.6 \
    stable/cert-manager
```

4. Follow the steps in the *restore guide* to restore your configuration.
5. Verify that your Issuers and Certificate resources are 'Ready':

```
$ kubectl get clusterissuer,issuer,certificates --all-namespaces
NAMESPACE      NAME                                  READY   SECRET
↪       AGE
cert-manager   cert-manager-webhook-ca               True    cert-manager-webhook-ca
↪       1m
```

(continues on next page)

(continued from previous page)

```
cert-manager    cert-manager-webhook-webhook-tls    True    cert-manager-webhook-
↪webhook-tls    1m
example-com     example-com-tls                     True    example-com-tls
↪        11s
```

Upgrading from v0.6 to v0.7

There are no special notes or considerations when upgrading from v0.6 to v0.7.

Upgrading from v0.7 to v0.8

Upgrading from v0.7 to v0.8 is possible using the regular *upgrade guide*.

All resources should continue to operate as before.

As part of v0.8, a new format **for configure ACME Certificate resources** has been introduced. Notably, challenge solver configuration has moved **from** the Certificate resource (under `certificate.spec.acme`) and now resides on your configure **Issuer** resource, under `issuer.spec.acme.solvers`.

This allows Certificate resources to be portable between different Issuer types.

Both the old and the new format of configuration are supported in the v0.8 release, so it is possible to **incrementally upgrade your resources** if you have a large, multi-team deployment of cert-manager that makes it complex to upgrade all manifests at once in place.

After upgrading, it is **strongly recommended** that you update your ACME Issuer and Certificate resources to the *new format*.

We will be removing support for the old format ahead of the 1.0 release.

The documentation has been updated to reflect configuring using the new format, and as such, exhaustive information can be found in the *Setting up ACME Issuers* document.

Performing an incremental switch to the new format

The following guide assumes you have 2 'solver types' currently in use across your cert-manager deployment - one for DNS01 and another for HTTP01 using an ingress class of `nginx`. The nginx based HTTP01 solver will be configured as the default solver type for Certificate resources that reference our issuer.

You can adjust the instructions below to fit your own configuration, either with more or less solvers as appropriate.

First, we will modify our ACME Issuer to add the new HTTP01 and DNS01 solvers. This operation **will not** effect any existing Certificates that already explicitly set a `certificate.spec.acme` field:

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

```

(continues on next page)

(continued from previous page)

```
    # The HTTP01 and DNS01 fields are now **deprecated**.
    # We leave them in place here so that any Certificates that still
    # specify a ``certificate.spec.acme`` stanza will continue to operate
    # correctly.
    # cert-manager will decide which configuration to use based on whether
    # the Certificate contains a ``certificate.spec.acme`` stanza.
    http01: {}
    dns01:
      providers:
      - name: cloudflare
        cloudflare:
          email: my-cloudflare-acc@example.com
          apiKeySecretRef:
            name: cloudflare-api-key-secret
            key: api-key

    # Configure the challenge solvers.
    solvers:
    # An empty selector will 'match' all Certificate resources that
    # reference this Issuer.
    - selector: {}
      http01:
        ingress:
          class: nginx
    - selector:
        # Any Certificate resources, or Ingress resources that use
        # ingress-shim and match the below label selector will use this
        # configured solver type instead of the default nginx based HTTP01
        # solver above.
        # You can continue to add new solver types if needed.
        # The most specific 'match' will be used.
        matchLabels:
          use-cloudflare-solver: "true"
      dns01:
        # Adjust the configuration below according to your environment.
        # You can view more example configurations for different DNS01
        # providers in the documentation: https://docs.cert-manager.io/en/latest/
↪tasks/issuers/setup-acme/dns01/index.html
        cloudflare:
          email: my-cloudflare-acc@example.com
          apiKeySecretRef:
            name: cloudflare-api-key-secret
            key: api-key
```

By retaining both the old and the new configuration format on the Issuer resource, we can begin the process of incrementally upgrading our Certificate resources.

Any Certificate resources that you have manually created (i.e. not managed by ingress-shim) must then be updated to remove the `certificate.spec.acme` stanza.

Given the above configuration, certificates will use the HTTP01 solver with the `nginx` ingress class in order to solve ACME challenges.

If a particular certificate requires a wildcard, or you simply want to use DNS01 for that certificate instead of HTTP01, you can add the `use-cloudflare-solver: "true"` label to your Certificate resources and the appropriate ACME challenge solver will be used.

Upgrading ingress-shim managed certificates to the new format

When using ingress-shim, cert-manager itself will create and manage your Certificate resource for you.

In order to support both the old and the new format simultaneously, ingress-shim will continue to set the `certificate.spec.acme` field on Certificate resources it manages.

In order to force ingress-shim to also use the new format, you must **remove** the old format configuration from your Issuer resources (i.e. `issuer.spec.acme.http01` and `issuer.spec.acme.dns01`).

When ingress-shim detects that these fields are not specified, it will clear/not set the `certificate.spec.acme` field.

If you are managing a certificate using ingress-shim that requires an alternative solver type (other than the default solver configured on the issuer which in this instance is the HTTP01 nginx solver), you can add labels to the Ingress resource which will be automatically copied across to the Certificate resource:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-test-ingress
  labels:
    use-cloudflare-solver: "true"
```

Confirming all Certificate resources are upgraded

In order to check if any of your Certificate resources still have the old configuration format, you can run the following command:

```
kubectl get certificate --all-namespaces \
  -o custom-columns="NAMESPACE:.metadata.namespace,NAME:.metadata.name,OWNER:.
↪metadata.ownerReferences[0].kind,OLD FORMAT:.spec.acme"

NAMESPACE   NAME    OWNER    OLD FORMAT
default     test    <none>   <none>
default     test2   Ingress  map[config:[map[domains:[abc.com]␣
↪http01:map[ingressClass:nginx]]]]
```

In the above example, we can see there are two Certificate resources.

The `test` resource has been updated to no longer include the `certificate.spec.acme` field.

The `test2` resource still specifies the old configuration format, however it **also** has an OwnerReference linking it to an **Ingress** resource. This is because the `test2` Certificate resource is managed by ingress-shim.

As mentioned in the previous section, ingress-shim managed certificates will only switch to the new format once the **old format** configuration on the **Issuer** resource has been removed. This means we need to continue to the next section in order to remove the old format configuration altogether from **Issuer** resource in order for ingress-shim to automatically migrate the `test2` Certificate resource.

Removing old configuration altogether

Once we've verified that all non-ingress-shim managed Certificate resources have been updated to not specify the `certificate.spec.acme` stanza using the command above, we can proceed to remove the `issuer.spec.acme.http01` and `issuer.spec.acme.dns01` stanzas from our Issuer resources. Once completed, the Issuer resource from the previous section should look like the following:

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

    # Configure the challenge solvers.
    solvers:
    # An empty selector will 'match' all Certificate resources that
    # reference this Issuer.
    - selector: {}
      http01:
        ingress:
          class: nginx
    - selector:
        # Any Certificate resources, or Ingress resources that use
        # ingress-shim and match the below label selector will use this
        # configured solver type instead of the default nginx based HTTP01
        # solver above.
        # You can continue to add new solver types if needed.
        # The most specific 'match' will be used.
        matchLabels:
          use-cloudflare-solver: "true"
      dns01:
        # Adjust the configuration below according to your environment.
        # You can view more example configurations for different DNS01
        # providers in the documentation: https://docs.cert-manager.io/en/latest/
↪tasks/issuers/setup-acme/dns01/index.html
        cloudflare:
          email: my-cloudflare-acc@example.com
          apiKeySecretRef:
            name: cloudflare-api-key-secret
            key: api-key
```

After applying the above Issuer resource, you should re-run the command from the last section to verify that the remaining ingress-shim managed Certificate resources have also been updated to the new format:

```
kubectl get certificate --all-namespaces \
  -o custom-columns="NAMESPACE:.metadata.namespace,NAME:.metadata.name,OWNER:.
↪metadata.ownerReferences[0].kind,OLD FORMAT:.spec.acme"

NAMESPACE   NAME    OWNER    OLD FORMAT
default     test    <none>   <none>
default     test2   Ingress  <none>
```

Manually triggering a Certificate to be issued to validate the full config

To be certain that you've correctly configured your new Issuer/Certificate resources, it is advised you attempt to issue a new Certificate after removing the old configuration format.

To do so, you can either:

- update the `secretName` field of an existing Certificate resource
- add an additional `dnsName` to one of your existing Certificate resources
- create a new Certificate resource

You should ensure that your Certificates are still be issued correctly to avoid any potential issues at renewal time.

Special notes for ingress-gce users

Users of the `ingress-gce` ingress controller may find that their experience configuring cert-manager to solve challenges using HTTP01 validation is slightly more painful using the new format, as it requires the `ingressName` field to be specified as a distinct `solver` on the Issuer resource (as opposed to in the past where the ingressName could be specified as a field on the `Certificate` resource).

This is a known issue, and a workaround is scheduled to be completed for v0.9.

In the meantime, ingress-gce users can either choose to manually create a new solver entry per Ingress resource they want to use to solve challenges, or otherwise continue to use the **old format** until a suitable alternative appears in v0.9.

Upgrading from v0.8 to v0.9

Due to a change in the API group that cert-manager deployments use (*apps/v1beta1* to *apps/v1*), cert-manager deployments must first be deleted before applying the new version. This will cause downtime until the new version has been applied. No data loss will occur during this operation however it is always advised to backup your data during an upgrade, which you can follow *here*. To perform this action run:

```
kubectl delete deployments --namespace cert-manager \
  cert-manager \
  cert-manager-cainjector \
  cert-manager-webhook
```

After this operation, follow the standard upgrade process as defined in the *upgrade guide*.

Upgrading from v0.9 to v0.10

Due to changes in the way the webhook component's TLS is bootstrapped in v0.10, you will need to delete your webhook's Certificate and Issuer resources.

If you are using a deployment tool that automatically handles this (i.e. Helm), there should be no additional action to take.

If you are using the 'static manifests' to install, you should run the following after upgrading:

```
kubectl delete -n cert-manager issuer cert-manager-webhook-ca cert-manager-webhook-
↪selfsign
kubectl delete -n cert-manager certificate cert-manager-webhook-ca cert-manager-
↪webhook-webhook-tls
kubectl delete apiservice v1beta1.admission.certmanager.k8s.io
```

The Secret resources used to contain TLS assets for the webhook are now automatically handled internally by cert-manager, so these resources are no longer required.

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:
    - http01:
        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 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
Venafi	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.

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.

```
1  apiVersion: certmanager.k8s.io/v1alpha1
2  kind: Certificate
3  metadata:
4    name: example
5  spec:
6    secretName: example-tls
7    duration: 24h
8    renewBefore: 12h
9    dnsNames:
10   - foo.example.com
11   - bar.example.com
12   issuerRef:
```

(continues on next page)

(continued from previous page)

```
    name: my-internal-ca
    kind: Issuer
```

4.1.2 Certificate Key Encoding

cert-manager Certificate resources support two types of key encodings for its private key known as the private key cryptography standards (PKCS). The two key encodings are PKCS#1 and PKCS#8.

The default encoding is PKCS#1, if the *keyEncoding* field of the Certificate spec is left empty.

A limitation exists where once a Certificate resource is generated with a specific key encoding, it cannot be generated with a different key encoding.

Example Usage

Here is an example of a Certificate specifying the use of PKCS#8 encoding on its private key.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example-pkcs8-cert
spec:
  secretName: example-pkcs8-secret
  keyEncoding: pkcs8
  dnsNames:
  - foo.example.com
  - bar.example.com
  issuerRef:
    name: my-internal-ca
    kind: Issuer
```

4.2 CertificateRequests

A 'CertificateRequest' is a resource in cert-manager that is used to request x509 certificates from an issuer. The resource contains a base64 encoded string of a PEM encoded certificate request which is sent to the referenced issuer. A successful issuance will return a signed certificate, based on the certificate signing request. 'CertificateRequets' are typically consumed and managed by controllers or other systems and should not be used by humans - unless specifically needed.

Note: To enable cert-manager's internal CertificateRequest controllers, supply the following feature gate: *–feature-gates=CertificateRequestControllers=true*

A simple CertificateRequest looks like the following:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: CertificateRequest
metadata:
  name: my-ca-cr
spec:
  csr:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQzNqQ0NBY1lDQVFBd2daZ3hDekFKQmdOVkJBWVRBbHBhTVI
```

(continues on next page)

(continued from previous page)

```
  isCA: false
  duraton: 90d
  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
    group: certmanager.k8s.io
```

This CertificateRequest will make cert-manager attempt to make the Issuer `letsencrypt-prod` in the default issuer pool `certmanager.k8s.io`, return a certificate based upon the certificate signing request. Other groups can be specified inside the `issuerRef` which will change the targeted issuers to other external, third party issuers you may have installed.

The resource also exposes the option for stating the certificate as CA and requested validity duration.

A successful issuance of the certificate signing request will cause an update to the resource, setting the status with the signed certificate, the CA of the certificate (if available), and setting the *Ready* condition to *True*.

Whether issuance of the controller was successful or not, a retry of the issuance will _not_ happen. It is the responsibility of some other controller to manage the logic and life cycle of CertificateRequets.

4.2.1 Conditions

CertificateRequests have a set of strongly defined conditions that should be used and relied upon by controllers or services to make decisions on what actions to take next on the resource. Each condition consists of the pair *Ready* - a boolean value, and *Reason* - a string. The set of values and meanings are as follows:

Ready	*Reason*	Condition Meaning
False	Pending	The CertificateRequest is currently pending, waiting for some other operation to take place. This could be that the Issuer does not exist yet or the Issuer is in the process of issuing a certificate.
False	Failed	The certificate has failed to be issued - either the returned certificate failed to be decoded or an instance of the referenced issuer used for signing failed. No further action will be taken on the CertificateRequest by it's controller.
True	Issued	A signed certificate has been successfully issued by the referenced Issuer.

4.3 Orders

Order resources are used by the ACME issuer to manage the lifecycle of an ACME 'order' for a signed TLS certificate.

When a Certificate resource is created that references an ACME issuer, cert-manager will create an Order resource in order to obtain a signed certificate.

As an end-user, you will never need to manually create an Order resource. Once created, an Order cannot be changed. Instead, a new Order resource must be created.

4.3.1 Debugging Order resources

In order to debug why a Certificate isn't being issued, we can first run `kubectl describe` on the Certificate resource we're having issues with:

```
$ kubectl describe certificate example-com

...
Events:
  Type    Reason        Age   From          Message
  ----    ------        ----  ----          -------
  Normal  Generated     1m    cert-manager  Generated new private key
  Normal  OrderCreated  1m    cert-manager  Created Order resource "example-com-
↪1217431265"
```

We can see here that Certificate controller has created an Order resource to request a new certificate from the ACME server.

Orders are a useful source of information when debugging failures issuing ACME certificates. By running `kubectl describe order` on a particular order, information can be gleaned about failures in the process:

```
$ kubectl describe order example-com-1248919344

...
Reason:
State:         pending
URL:           https://acme-v02.api.letsencrypt.org/acme/order/41123272/265506123
Events:
  Type    Reason   Age   From          Message
  ----    ------   ----  ----          -------
  Normal  Created  1m    cert-manager  Created Challenge resource "example-com-
↪1217431265-0" for domain "test1.example.com"
  Normal  Created  1m    cert-manager  Created Challenge resource "example-com-
↪1217431265-1" for domain "test2.example.com"
```

Here we can see that cert-manager has created two Challenge resources in order to fulfil the requirements of the ACME order to obtain a signed certificate.

You can then go on to run `kubectl describe challenge example-com-1217431265-0` to further debug the progress of the Order.

Once an Order is successful, you should see an event like the following:

```
$ kubectl describe order example-com-1248919344

...
Reason:
State:         valid
URL:           https://acme-v02.api.letsencrypt.org/acme/order/41123272/265506123
Events:
  Type    Reason      Age   From          Message
  ----    ------      ----  ----          -------
  Normal  Created     72s   cert-manager  Created Challenge resource "example-com-
↪1217431265-0" for domain "test1.example.com"
  Normal  Created     72s   cert-manager  Created Challenge resource "example-com-
↪1217431265-1" for domain "test2.example.com"
  Normal  OrderValid  4s    cert-manager  Order completed successfully
```

If the Order is not completing successfully, you can debug the challenges for the Order by running `kubectl describe` on the Challenge resource.

For more information on debugging Challenge resources, read the *challenge reference docs*.

4.4 Challenges

Challenge resources are used by the ACME issuer to manage the lifecycle of an ACME 'challenge' that must be completed in order to complete an 'authorization' for a single DNS name/identifier.

When an **Order** resource is created, the order controller will create Challenge resources for each DNS name that is being authorized with the ACME server.

As an end-user, you will never need to manually create a Challenge resource. Once created, a Challenge cannot be changed. Instead, a new Challenge resource must be created.

4.4.1 Challenge lifecycle

After a Challenge resource has been created, it will be initially queued for processing. Processing will not begin until the challenge has been 'scheduled' to start. This scheduling process prevents too many challenges being attempted at once, or multiple challenges for the same DNS name being attempted at once. For more information on how challenges are scheduled, read the *challenge scheduling* section.

Once a challenge has been scheduled, it will first be 'synced' with the ACME server in order to determine its current state. If the challenge is already valid, its 'state' will be updated to 'valid', and also set `status.processing = false` to 'unschedule' itself.

If the challenge is still 'pending', the challenge controller will 'present' the challenge using the configured solver, one of HTTP01 or DNS01. Once the challenge has been 'presented', it will set `status.presented=true`.

Once 'presented', the challenge controller will perform a 'self check' to ensure that the challenge has 'propagated' (i.e. the authoritve DNS servers have been updated to respond correctly, or the changes to the ingress resources have been observed and in-use by the ingress controller).

If the self check fails, cert-manager will retry the self check with a fixed 10 second retry interval. Challenges that do not ever complete the self check will continue retrying until the user intervenes.

Once the self check is passing, the ACME 'authorization' associated with this challenge will be 'accepted' (TODO: add link to accepting challenges section of ACME spec).

The final state of the authorization after accepting it will be copied across to the Challenge's `status.state` field, as well as the 'error reason' if an error occurred whilst the ACME server attempted to validate the challenge.

Once a Challenge has entered the `valid`, `invalid`, `expired` or `revoked` state, it will set `status.processing=false` to prevent any further processing of the ACME challenge, and to allow another challenge to be scheduled if there is a backlog of challenges to complete.

4.4.2 Challenge scheduling

Instead of attempting to process all challenges at once, challenges are 'scheduled' by cert-manager.

This scheduler applies a cap on the maximum number of simultaneous challenges as well as disallows two challenges for the same DNS name and solver type (http-01 or dns-01) to be completed at once.

The maximum number of challenges that can be processed at a time is 60 as of ddff78.

4.4.3 Debugging Challenge resources

In order to determine why an ACME Certificate is not being issued, we can debug using the 'Challenge' resources that cert-manager has created.

In order to determine which Challenge is failing, you can run `kubectl get challenges`:

```
$ kubectl get challenges

NAME                          STATE     DOMAIN             REASON                                                 ␣
↪           AGE
example-com-1217431265-0  pending   example.com        Waiting for dns-01 challenge␣
↪propagation   22s
```

This shows that the challenge has been presented using the DNS01 solver successfully and now cert-manager is waiting for the 'self check' to pass.

You can get more information about the challenge by using `kubectl describe`:

```
$ kubectl describe challenge example-com-1217431265-0

...
Status:
  Presented:   true
  Processing:  true
  Reason:      Waiting for dns-01 challenge propagation
  State:       pending
Events:
  Type    Reason     Age   From          Message
  ----    ------     ----  ----          -------
  Normal  Started    19s   cert-manager  Challenge scheduled for processing
  Normal  Presented  16s   cert-manager  Presented challenge using dns-01 challenge␣
↪mechanism
```

Progress about the state of each challenge will be recorded either as Events or on the Challenge's `status` block (as shown above).

4.4.4 Troubleshooting failing challenges

Todo: add section describing common issues and resolutions when challenges are failing

4.5 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:

```
1  apiVersion: certmanager.k8s.io/v1alpha1
2  kind: Issuer
3  metadata:
4    name: letsencrypt-prod
5    namespace: edge-services
6  spec:
7    acme:
8      # The ACME server URL
9      server: https://acme-v02.api.letsencrypt.org/directory
10     # Email address used for ACME registration
```

(continues on next page)

(continued from previous page)

```
    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*.

4.5.1 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.

4.5.2 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.

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.

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.

4.5.3 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
Venafi	Supports issuing certificates from Venafi Cloud & TPP

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.

4.6 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:

```yaml
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.

4.7 cainjector controller

4.8 API documentation

CHAPTER 5

Design and Proposals

To view design documents please visit the link here.

CHAPTER 6

Development documentation

6.1 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.

6.1.1 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
# If using Minikube v0.25.0 or older:
$ minikube start --extra-config=apiserver.Authorization.Mode=RBAC
# Otherwise:
$ minikube start

# 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
```

6.1.2 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 and MacOS; Windows may require further changes.

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

- Git
- Mercurial

You can then run `./hack/update-vendor.sh` to regenerate any dependencies, and `make build` to build the docker images.

6.1.3 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:canary
```

6.1.4 Deploy that version with helm

```
# Install custom resources before running helm
$ kubectl apply -f deploy/manifests/00-crds.yaml

# IMPORTANT: if you are deploying into a namespace that **already exists**,
# you MUST ensure the namespace has an additional label on it in order for
# the deployment to succeed
$ kubectl label namespace <deployment-namespace> certmanager.k8s.io/disable-
↪validation="true"

# Install our freshly built cert-manager image
$ helm install \
    --set image.tag=canary \
    --set image.pullPolicy=Never \
    --set cainjector.image.tag=canary \
    --set cainjector.pullPolicy=Never \
    --set webhook.image.tag=canary \
    --set webhook.pullPolicy=Never \
    --name cert-manager \
    ./deploy/charts/cert-manager
```

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

6.1.5 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
```

6.2 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.

6.2.1 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.

6.2.2 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.

6.3 Contributing DNS01 providers

6.3.1 WARNING

Because of the overwhelming number of PRs for new DNS providers, We're changing how we handle the DNS01 contributions. See this post on the mailing list for more information.

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/tasks/acme/configuring-dns01/`. 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.

6.4 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

(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.
```

6.5 Release process

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

6.5.1 Minor releases

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

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.

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/tasks/upgrading
2. Ensure all strings of versions have been updated:
    - deploy/charts/cert-manager/README.md
    - docs/getting-started/install/kubernetes.rst
    - docs/getting-started/install/openshift.rst
    - docs/getting-started/webhook.rst
    - docs/tutorials/acme/quick-start/index.rst
3. Create a new release branch (e.g. `release-0.5`)
4. Push it to the `jetstack/cert-manager` repository
5. Gather release notes since the previous release:
    - Download, install and run the latest version of release-notes:

```
* $ go get k8s.io/release; go install $GOPATH/src/k8s.io/release/cmd/release-notes/.
* $ mkdir -p design/release-notes/release-*X.Y*
* $ export GITHUB_TOKEN=*your-token*
* $ $GOPATH/bin/release-notes -release-version v*X.Y* -github-repo cert-manager -
↪github-org jetstack -requiredAuthor "" -start-sha=$(git rev-parse *X.Y-1.0*) -end-
↪sha=$(git rev-parse HEAD) -output design/release-notes/release-*X.Y*/draft-release-
↪notes.md
* # Add additional blurb, notable items and characterise Changelog.
```

Finally, create a new tag taken from the release branch, e.g. `v0.5.0`.

6.5.2 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.

Release schedule

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

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. Ensure all strings of versions have been updated:
    - deploy/charts/cert-manager/README.md
    - docs/getting-started/install/kubernetes.rst
    - docs/getting-started/install/openshift.rst
    - docs/getting-started/webhook.rst
    - docs/tutorials/acme/quick-start/index.rst
2. Iterate on review feedback (hopefully this will be minimal) and submit changes to `master` of cert-manager, performing a rebase of release-x.y.
3. Gather release notes since the previous release:

```
* $ go get k8s.io/release; go install $GOPATH/src/k8s.io/release/cmd/release-notes/.
* $ mkdir -p design/release-notes/release-*X.Y*
* $ export GITHUB_TOKEN=*your-token*
* $ $GOPATH/bin/release-notes -release-version v*X.Y* -github-repo cert-manager -
↪github-org jetstack -requiredAuthor "" -start-sha=$(git rev-parse *X.Y.Z-1*) -end-
↪sha=$(git rev-parse release-*X.Y*) -output design/release-notes/release-*X.Y*/draft-
↪release-notes-*Z*.md
* # Add additional blurb, notable items and characterise Changelog.
```

Finally, create a new tag taken from the release branch, e.g. `v0.5.1`.

6.6 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.

6.6.1 Installation instructions

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

```
pip install --user -r requirements.txt
```

6.6.2 Generating documentation locally

You can generate the documentation locally with the following command:

```
make html
```

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

```
open _build/html/index.html
```

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.