cert-manager Documentation

Jetstack Ltd

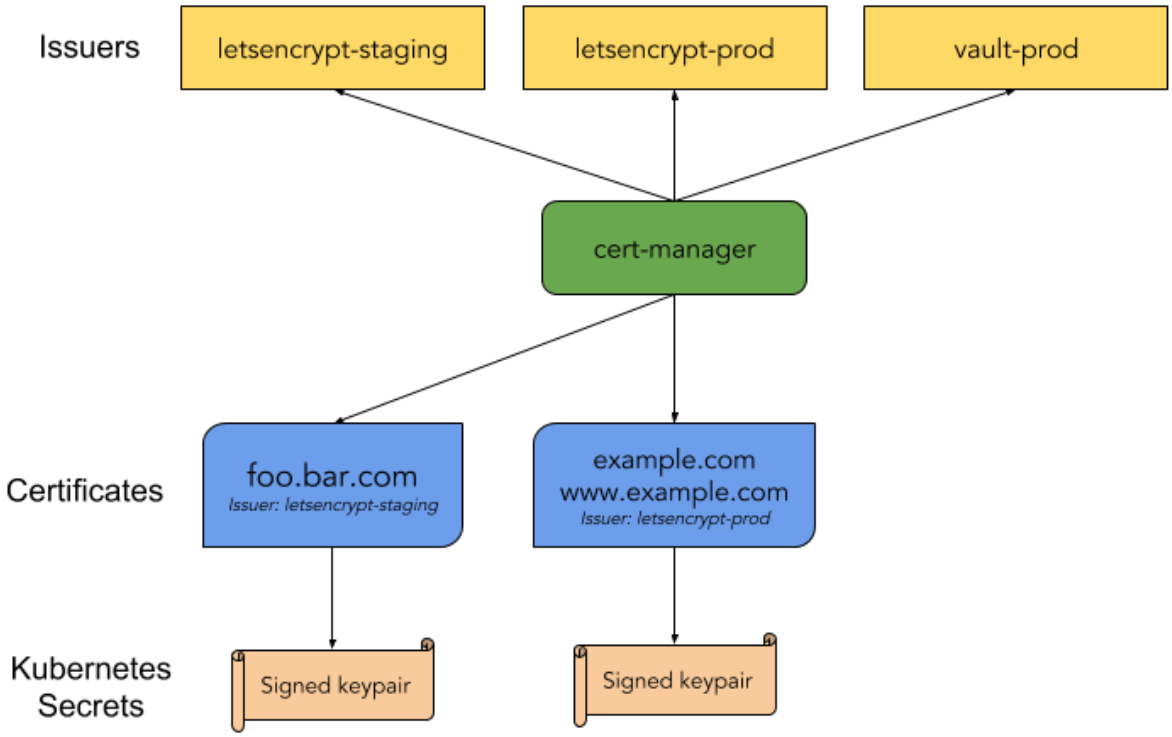
Jul 20, 2018

Contents:

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

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

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



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

CHAPTER 1

Getting started

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

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

1.1 1. Configuring Helm and Tiller

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

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

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

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

1.2 2. Installing cert-manager

1.2.1 With Helm

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

You can install the chart with the following command:

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

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

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

1.2.2 With static manifests

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

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

1.3 3. Configuring your first Issuer or ClusterIssuer

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

These represent a certificate authority from which signed x509 certificates can be obtained, such as Let's Encrypt, or your own signing key pair stored in a Kubernetes Secret resource.

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

On the other hand, a *ClusterIssuer* is a cluster wide version of an *Issuer*. It is able to be referenced by *Certificate* resources in any namespace. Users often create `letsencrypt-staging` and `letsencrypt-prod` *ClusterIssuers* if they operate a single-tenant environment and want to expose a cluster-wide mechanism for obtaining TLS certificates from Let's Encrypt.

CHAPTER 2

Tutorials

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

2.1 ACME Issuer Tutorials

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

2.1.1 Migrating from kube-lego

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

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

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

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

By the end of the guide, we should have:

1. Scaled down and removed kube-lego
2. Installed cert-manager

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

2.1.1.1 1. Scale down kube-lego

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

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

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

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

2.1.1.2 2. Deploy cert-manager

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

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

2.1.1.3 3. Obtaining your ACME account private key

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We then submit this file to our Kubernetes cluster:

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

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

```
$ kubectl describe clusterissuer letsencrypt-staging
Name:         letsencrypt-staging
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  certmanager.k8s.io/v1alpha1
```

(continues on next page)

(continued from previous page)

```
Kind:         ClusterIssuer
Metadata:
  Cluster Name:
  Creation Timestamp:  2017-11-30T22:33:40Z
  Generation:          0
  Resource Version:    4450170
  Self Link:           /apis/certmanager.k8s.io/v1alpha1/letsencrypt-staging
  UID:                 83d04e6b-d61e-11e7-ac26-42010a840044
Spec:
  Acme:
    Email:  user@example.com
    Http 01:
    Private Key Secret Ref:
      Key:
      Name:  letsencrypt-private-key
    Server:  https://acme-staging-v02.api.letsencrypt.org/directory
Status:
  Acme:
    Uri:  https://acme-staging-v02.api.letsencrypt.org/acme/acct/11217539
  Conditions:
    Last Transition Time:  2018-04-12T17:32:30Z
    Message:               The ACME account was registered with the ACME server
    Reason:                ACMEAccountRegistered
    Status:                True
    Type:                  Ready
```

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

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

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

```
helm upgrade cert-manager \
    stable/cert-manager \
    --namespace kube-system \
    --set ingressShim.defaultIssuerName=letsencrypt-staging \
    --set ingressShim.defaultIssuerKind=ClusterIssuer
```

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

2.1.1.6 6. Verify each ingress now has a corresponding Certificate

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

You should be able to check this by running:

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

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

We can also verify that cert-manager has 'adopted' the old TLS certificates by 'describing' one of these newly created certificates:

```
$ kubectl describe certificate my-example-certificate
...
Events:
  Type    Reason            Age                From                     Message
  ----    ------            ----               ----                     -------
  Normal  RenewalScheduled  1m                 cert-manager-controller  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.

2.1.2 Securing nginx-ingress with Let's Encrypt

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

2.1.2.1 Prerequisites

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

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

2.1.3 Issuing an ACME certificate using DNS validation

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

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

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

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

```
1  apiVersion: certmanager.k8s.io/v1alpha1
2  kind: Issuer
3  metadata:
4    name: letsencrypt-staging
5    namespace: default
6  spec:
7    acme:
8      server: https://acme-staging-v02.api.letsencrypt.org/directory
9      email: user@example.com
10
11     # Name of a secret used to store the ACME account private key
```

(continues on next page)

(continued from previous page)

```
    privateKeySecretRef:
      name: letsencrypt-staging

    # ACME DNS-01 provider configurations
    dns01:

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

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

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

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

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

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

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

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

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  issuerRef:
    name: letsencrypt-staging
  commonName: '*.example.com'
  dnsNames:
  - example.com
```

(continues on next page)

(continued from previous page)

```
  - 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'* 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.

2.1.4 Issuing an ACME certificate using HTTP validation

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

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

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

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

```
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.

2.2 CA Issuer Tutorials

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

2.2.1 Creating a simple CA based issuer

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

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

2.2.1.1 1. (Optional) Generate a signing key pair

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

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

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

(continues on next page)

(continued from previous page)

```
# 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.

2.2.1.2 2. Save the signing key pair as a Secret

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

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

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

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

2.2.1.3 3. Creating an Issuer referencing the Secret

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

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

We are now ready to obtain certificates!

2.2.1.4 4. Obtain a signed Certificate

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

```
1 apiVersion: certmanager.k8s.io/v1alpha1
2 kind: Certificate
3 metadata:
4   name: example-com
5   namespace: default
6 spec:
7   secretName: example-com-tls
8   issuerRef:
```

(continues on next page)

(continued from previous page)

```
    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
  dnsNames:
  - example.com
  - www.example.com
```

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

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

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

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

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

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

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

2.3 Vault Issuer Tutorials

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

2.3.1 Vault Installation

2.3.1.1 Installing Vault

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

2.3.1.2 Vault PKI Backend

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

2.3.2 Vault Authentication with a AppRole

This Vault authentication method uses a Vault AppRole.

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

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

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

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

We can now create a cluster issuer referencing this secret:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: vault-issuer
  namespace: default
spec:
  vault:
    path: pki_int/sign/example-dot-com
    server: https://vault
    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*.

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

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

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

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

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

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

2.3.3 Vault Authentication with a Token

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

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

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

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

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

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

We can now create an issuer referencing this secret:

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

(continues on next page)

(continued from previous page)

```
  namespace: default
spec:
  vault:
    auth:
      tokenSecretRef:
        name: cert-manager-vault-token
        key: token
    path: pki_int/sign/example-dot-com
    server: https://vault
```

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.

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*.

CHAPTER 3

Administrative tasks

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

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

3.1 Upgrading cert-manager

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

3.1.1 Upgrading from v0.2 to v0.3

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

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

A brief summary:

- Supporting resources for ClusterIssuers (e.g. signing CA certificates, or ACME account private keys) will now be stored in the same namespace as cert-manager, instead of kube-system in previous versions (#329, @munnerz)
- Switch to ConfigMaps instead of Endpoints for leader election (#327, @mikebryant)
- Removing support for ACMEv1 in favour of ACMEv2 (#309, @munnerz)
- Removing ingress-shim and compiling it into cert-manager itself (#502, @munnerz)
- Change to the default behaviour of ingress-shim. It now generates Certificates with the `ingressClass` field set instead of the `ingress` field. This will mean users of ingress controllers that assign a single IP to a single

Ingress (e.g. the GCE ingress controller) will no longer work without adding a new annotation to your ingress resource.

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

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

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

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

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

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

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

3.1.1.2 Switch to ConfigMaps instead of Endpoints for leader election

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

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

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

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

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

3.1.1.3 Removing support for ACMEv1 in favour of ACMEv2

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

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

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

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

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

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

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

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

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

3.1.1.5 Change to the default behaviour of ingress-shim

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

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

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

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

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

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

CHAPTER 4

Reference documentation

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

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

4.1 Certificates

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

A simple Certificate could be defined as:

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

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

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

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

4.1.1 ACME Specific Certificate Config

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

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

Todo: Make this document more prominent

4.2 Issuers

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

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

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

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

4.2.1 Namespacing

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

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

4.2.2 Ambient Credentials

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

4.2.2.1 Example Usage

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

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

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

4.2.2.2 When are Ambient Credentials used

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

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

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

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

4.2.3 Supported Issuer types

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

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

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

4.2.3.1 ACME Configuration

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

4.2.3.1.1 HTTP01 Challenge Provider

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

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

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

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

4.2.3.1.2 DNS01 Challenge Provider

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

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

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

Setting nameservers for DNS01 self check

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

Example usage:

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

Supported DNS01 providers

A number of different DNS providers are supported for the ACME issuer. Below is a listing of them all, with an example block of configuration:

Google CloudDNS

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

Amazon Route53

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

(continues on next page)

(continued from previous page)

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

Cert-manager requires the following IAM policy.

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

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

Cloudflare

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

Akamai FastDNS

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

(continues on next page)

(continued from previous page)

```
    key: clientSecret
  accessTokenSecretRef:
    name: akamai-dns
    key: accessToken
```

4.2.3.2 CA Configuration

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

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

Todo: Expand out CA Issuer reference documentation

4.2.3.3 Vault Configuration

Vault Issuers issue certificates from Hashicorp's Vault.

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

Todo: Expand out Vault Issuer reference documentation

4.2.3.4 Self-signed Configuration

Self signed Issuers will issue self signed certificates.

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

4.3 ClusterIssuers

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

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

The docs for Issuer resources apply equally to ClusterIssuers.

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

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

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

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

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

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

4.4 ingress-shim

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

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

4.4.1 How it works

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

As of the time of writing, it **will not** update Certificate resources if your Ingress resource changes. It is up to yourself to ensure the corresponding Certificate resource is as required.

4.4.2 Configuration

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

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

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

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

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

4.4.3 Supported annotations

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

- `certmanager.k8s.io/issuer` - the name of an Issuer to acquire the certificate required for this ingress from. The Issuer **must** be in the same namespace as the Ingress resource.
- `certmanager.k8s.io/cluster-issuer` - the name of a ClusterIssuer to acquire the certificate required for this ingress from. It does not matter which namespace your Ingress resides, as ClusterIssuers are non-namespaced resources.
- `certmanager.k8s.io/acme-challenge-type` - by default, if the Issuer specified is an ACME issuer (either through ingress-shim's defaults, or with one of the above annotations), the ingress-shim will set the ACME challenge mechanism on the Certificate resource it creates to 'http01'. This annotation can be used to alter this behaviour. Must be one of 'http01' or 'dns01'.
- `certmanager.k8s.io/acme-dns01-provider` - if the ACME challenge type has been set to dns01, this annotation **must** be specified to instruct cert-manager which DNS provider (as configured on the specified Issuer resource) should be used. This field is required if the challenge type is set to DNS01.
- `kubernetes.io/tls-acme: "true"` - this annotation requires additional configuration of the ingress-shim (see above). Namely, a default issuer must be specified as arguments to the ingress-shim container.
- `certmanager.k8s.io/acme-http01-edit-in-place: "true"` - if the ACME challenge type has been set to http01, and the ingress has the 'kubernetes.io/tls-acme: true' annotation, this controls whether the ingress is modified 'in-place', or a new one created specifically for the http01 challenge. If present, and set to "true" the existing ingress will be modified. Any other value, or the absence of the annotation assumes "false".

4.5 API documentation

CHAPTER 5

Development documentation

5.1 Develop with minikube

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

5.1.1 Start minikube

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

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

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

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

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

# Install helm
$ helm init
```

5.1.2 Build a dev version of cert-manager

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

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

5.1.3 Deploy that version with helm

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

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

5.1.4 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
```