Getting Started with the Enterprise Helm Charts

Introduction

This document will walk you through how to get started with our Element Server Suite Helm Charts. These charts are provided to be used in environments which typically deploy applications by helm charts. If you are unfamiliar with helm charts, we'd highly recommend that you start with our Enterprise Installer.

General concepts

ESS deployment rely on the following components to deploy the workloads on a kubernetes cluster :

1. Updater : It reads an ElementDeployment CRD manifest, and generates the associated individual Element CRDs manifests linked together
2. Operator : It reads the individual Element CRDs manifests to generates the associated kubernetes workloads
3. ElementDeployment : This CRD is a simple structure following the pattern :

```
spec:
  global:
    k8s:
      # Global settings that will be applied by default to all workloads if not forced locally. This is where you will be able to configure a default ingress certificate, default number of replicas on the deployments, etc.
    config:
      # Global configuration that can be used by every element component
    secretName: # The global secret name. Required secrets keys can be found in the description of this field using `kubectl explain`. Every config named `<foo>SecretKey` will point to a secret key containing the secret targetted by this secret name.
  components:
    <component name>:
      k8s:
```

```
    # Local kubernetes configuration of this component. You can override here the global values to force a certain behaviour for each components.
    config:
      # This component configuration
    secretName: # The component secret name containing secret values. Required secrets keys can be found in the description of this field using `kubectl explain`.  Every config named `<foo>SecretKey` will point to a secret key containing the secret targetted by this secret name.
  <another component>:
    ...
```

Any change to the ElementDeployment manifest deployed in the namespace will trigger a reconciliation loop. This loop will update the Element manifests read by the Operator. It will again trigger a reconciliation loop in the Operator process, which will update kubernetes workloads accordingly.

If you manually change a workload, it will trigger a reconciliation loop and the Operator will override your change on the workload.

The deployment must be managed only through the ElementDeployment CRD.

Installing the Operator and the Updater helm charts

We advise you to deploy the helm charts in one of the deployments model :

1. Cluster-Wide deployment : In this mode, the CRDs Conversion Webhook and the controller managers are deployed in their own namespace, separated from ESS deployments. They are able to manage ESS deployments in any namespace of the cluster The install and the upgrade of the helm chart requires cluster admin permissions.
2. Namespace-scoped deployment : In this mode, only the CRDs conversion webhooks require cluster admin permissions. The Controller managers are deployed directly in the namespace of the element deployment. The install and the upgrade of ESS does not require cluster admin permissions if the CRDs do not change.

All-in-one deployment (Requires cert-manager)

When cert-manager is present in the cluster, it is possible to use the all-in-one `ess-system` helm chart to deploy the operator and the updater.

First, let's add the ess-system repository to helm, replace `ems_image_store_username` and `ems_image_store_token` with the values provided to you by Element.

```
helm repo add ess-system https://registry.element.io/helm/ess-system --username
<ems_image_store_username> --password '<ems_image_store_token>'
```

Cluster-wide deployment

When deploying ESS-System as a cluster-wide deployment, updating ESS requires ClusterAdmin permissions.

Create the following values file :

```
emsImageStore:
  username: <username>
  password: <password>

element-operator:
  clusterDeployment: true
  deployCrds: true  # Deploys the CRDs and the Conversion Webhooks
  deployCrdRoles: true  # Deploys roles to give permissions to users to manage specific ESS CRs
  deployManager: true  # Deploys the controller managers

element-updater:
  clusterDeployment: true
  deployCrds: true  # Deploys the CRDs and the Conversion Webhooks
  deployCrdRoles: true  # Deploys roles to give permissions to users to manage specific ESS CRs
  deployManager: true  # Deploys the controller managers
```

Namespace-scoped deployment

When deploying ESS-System as a namespace-scoped deployment, you have to deploy `ess-system` in two parts :

1. One for the CRDs and the conversion webhooks. This part will be managed with ClusterAdmin permissions. These update less often.
2. One for the controller managers. This part will be managed with namespace-scoped permissions.

In this mode, the `ElementDeployment` CR is deployed in the same namespace as the controller-managers.

Create the following values file to deploy the CRDs and the conversion webhooks :

```
emsImageStore:
  username: <username>
  password: <password>

element-operator:
  clusterDeployment: true
  deployCrds: true  # Deploys the CRDs and the Conversion Webhooks
  deployCrdRoles: false  # Deploys roles to give permissions to users to manage specific ESS CRs
  deployManager: false  # Deploys the controller managers

element-updater:
  clusterDeployment: true
  deployCrds: true  # Deploys the CRDs and the Conversion Webhooks
  deployCrdRoles: false  # Deploys roles to give permissions to users to manage specific ESS CRs
  deployManager: false  # Deploys the controller managers
```

Create the following values file to deploy the controller managers in their namespace :

```
emsImageStore:
  username: <username>
  password: <password>

element-operator:
  clusterDeployment: false
  deployCrds: false  # Deploys the CRDs and the Conversion Webhooks
  deployCrdRoles: false  # Deploys roles to give permissions to users to manage specific ESS CRs
  deployManager: true  # Deploys the controller managers

element-updater:
  clusterDeployment: false
  deployCrds: false  # Deploys the CRDs and the Conversion Webhooks
  deployCrdRoles: false  # Deploys roles to give permissions to users to manage specific ESS CRs
  deployManager: true  # Deploys the controller managers
```

Without cert-manager present on the cluster

First, let's add the element-updater and element-operator repositories to helm, replace `ems_image_store_username` and `ems_image_store_token` with the values provided to you by Element.

```
helm repo add element-updater https://registry.element.io/helm/element-updater --username
<ems_image_store_username> --password '<ems_image_store_token>'
helm repo add element-operator https://registry.element.io/helm/element-operator --username
<ems_image_store_username> --password '<ems_image_store_token>'
```

Now that we have the repositories configured, we can verify this by:

```
helm repo list
```

and should see the following in that output:

```
NAME                URL
element-operator    https://registry.element.io/helm/element-operator
element-updater     https://registry.element.io/helm/element-updater
```

N.B. This guide assumes that you are using the `element-updater` and `element-operator` namespaces. You can call it whatever you want and if it doesn't exist yet, you can create it with: `kubectl create ns <name>`.

Generating an image pull secret with EMS credentials

To generate an `ems-credentials` to be used by your helm chart deployment, you will need to generate an authentication token and palce it in a secret.

```
kubectl create secret -n element-updater docker-registry ems-credentials --docker-server=registry.element.io --
docker-username=<EMSusername> --docker-password=<EMStoken>`
kubectl create secret -n element-operator docker-registry ems-credentials --docker-server=registry.element.io --
docker-username=<EMSusername> --docker-password=<EMStoken>`
```

Generating a TLS secret for the webhook

The conversion webhooks need their own self-signed CA and TLS certificate to be integrated into kubernetes.

For example using `easy-rsa` :

```
easyrsa init-pki
easyrsa --batch "--req-cn=ESS-CA`date +%s`" build-ca nopass
easyrsa --subject-alt-name="DNS:element-operator-conversion-webhook.element-operator"\
  --days=10000 \
  build-server-full element-operator-conversion-webhook nopass
easyrsa --subject-alt-name="DNS:element-updater-conversion-webhook.element-updater"\
  --days=10000 \
  build-server-full element-updater-conversion-webhook nopass
```

Create a secret for each of these two certificates :

```
kubectl create secret tls element-operator-conversion-webhook --cert=pki/issued/element-operator-conversion-webhook.crt --key=pki/private/element-operator-conversion-webhook.key  --namespace element-operator
kubectl create secret tls element-updater-conversion-webhook --cert=pki/issued/element-updater-conversion-webhook.crt --key=pki/private/element-updater-conversion-webhook.key  --namespace element-updater
```

Installing the helm chart for the `element-updater` and the `element-operator`

Create the following values file to deploy the controller managers in their namespace :

`values.element-operator.yml` :

```
clusterDeployment: true
deployCrds: true  # Deploys the CRDs and the Conversion Webhooks
deployCrdRoles: true  # Deploys roles to give permissions to users to manage specific ESS CRs
deployManager: true  # Deploys the controller managers
crds:
  conversionWebhook:
    caBundle: # Paste here the content of `base64 pki/ca.crt -w 0`
    tlsSecretName: element-operator-conversion-webhook
    imagePullSecret: ems-credentials
operator:
  imagePullSecret: ems-credentials
```

`values.element-updater.yml` :

```
clusterDeployment: true
deployCrds: true  # Deploys the CRDs and the Conversion Webhooks
deployCrdRoles: true  # Deploys roles to give permissions to users to manage specific ESS CRs
```

```
  deployManager: true  # Deploys the controller managers
  crds:
    conversionWebhook:
      caBundle: # Paste here the content of `base64 pki/ca.crt -w 0`
      tlsSecretName: element-updater-conversion-webhook
  updater:
    imagePullSecret: ems-credentials
```

Run the helm install command :

```
helm install element-operator element-operator/element-operator --namespace element-operator -f values.yaml
helm install element-updater element-updater/element-updater --namespace element-updater -f values.yaml
```

Now at this point, you should have the following 4 containers up and running:

```
[user@helm ~]$ kubectl get pods -n element-operator
NAMESPACE          NAME                                          READY   STATUS    RESTARTS      AGE
element-operator     element-operator-controller-manager-c8fc5c47-nzt2t    2/2     Running   0           6m5s
element-operator     element-operator-conversion-webhook-7477d98c9b-xc89s   1/1     Running   0
6m5s
[user@helm ~]$ kubectl get pods -n element-updater
NAMESPACE          NAME                                          READY   STATUS    RESTARTS      AGE
element-updater      element-updater-controller-manager-6f8476f6cb-74nx5    2/2     Running   0           106s
element-updater      element-updater-conversion-webhook-65ddcbb569-qzbfs    1/1     Running   0           81s
```

Generating the ElementDeployment CR to Deploy Element Server Suite

Using the ess-stack helm-chart

The `ess-stack` helm chart is available in the `ess-system` repository :

```
helm repo add ess-system https://registry.element.io/helm/ess-system --username
<ems_image_store_username> --password '<ems_image_store_token>'
```

It will deploy an ElementDeployment CR and its associated secrets from the chart values file.

The values file will contain the following structure :

- Available Components & Global settings can be found under https://ess-schemas-docs.element.io
- For each `SecretKey` variable, the value will point to a secret key under `secrets`. For example, `components.synapse.config.macaroonSecretKey` is `macaroon`, so a `macaroon` secret must exists under `secrets.synapse.content`.

```
emsImageStore:
  username: <username>
  password: <password>

secrets:
  global:
    content:
      genericSharedSecret: # generic shared secret
  synapse:
    content:
        macaroon: # macaroon
        postgresPassword: # postgres password
        registrationSharedSecret: # registration shared secret

 # globalOptions contains the global properties of the ELementDeployment CRD
globalOptions:
  config:
    domainName: # your base domain
  k8s:
    ingresses:
      tls:
        mode: certmanager
        certmanager:
          issuer: letsencrypt
    workloads:
      replicas: 1

components:
```

```
  elementWeb:
    k8s:
      ingress:
        fqdn:  # element web fqdn
  synapse:
    config:
      media:
        volume:
          size: 5Gi
      postgresql:
        database: # postgres database
        host:  # postgres host
        port: 5432
        user: # postgres user
    k8s:
      ingress:
        fqdn: # synapse fqdn
  wellKnownDelegation:
    config: {}
    k8s: {}
```

Writing your own ElementDeployment CR

Here is a small sample to deploy the basic components using your own certificate files. This is provided as an example, as ElementDeployment supports a whole range of configuration option that you can explore in :

- The documentation website at https://ess-schemas-docs.element.io
- the GUI
- through `kubectl explain` command : `kubectl explain elementdeployment.matrix.element.io.spec.components`

```
apiVersion: matrix.element.io/v1alpha1
kind: ElementDeployment
metadata:
  name: <element_deployment_name>
  namespace: <target namespace>
spec:
  global:
```

```
  k8s:
    ingresses:
      ingressClassName: "public"
    workloads:
      dockerSecrets:
      - name: dockerhub
        url: docker.io
      - name: element-registry
        url: registry.element.io
    storage:
      storageClassName: "standard"
  secretName: global
  config:
    genericSharedSecretSecretKey: genericSharedSecret
    domainName: "deployment.tld"
components:
  elementWeb:
    secretName: external-elementweb-secrets
    k8s:
      ingress:
        tls:
          mode: certfile
          certificate:
            certFileSecretKey: eleweb.tls
            privateKeySecretKey: eleweb.crt
        fqdn: element-web.tld
  synapse:
    secretName: external-synapse-secrets
    config:
      maxMauUsers: 100
      media:
        volume:
          size: 1
      postgresql:
        host: "<postgresql server>"
        user: "<user>"
        database: "<db>"
        passwordSecretKey: pgpassword
        sslMode: disable
    k8s:
```

```
    ingress:
      fqdn: synapse.tld
  tls:
    mode: certfile
    certificate:
      certFileSecretKey: synapse.tls
      privateKeySecretKey: synapse.crt
wellKnownDelegation:
  secretName: external-wellknowndelegation-secrets
  k8s:
    ingress:
      tls:
        mode: certfile
        certificate:
          certFileSecretKey: wellknown.tls
          privateKeySecretKey: wellknown.crt
```

To inject secret values in the CR, you will have to create the following secrets :

- `name: global` with data key `genericSharedSecret` containing any random value. It will be used as a seed for all secrets generated by the updater.
- `name: external-elementweb-secrets` with data keys `eleweb.tls` containing element web private key and `eleweb.crt` containing element web certificate.
- `name: external-synapse-secrets` with data keys `synapse.tls` containing synapse private key and `synapse.crt` containing synapse certificate. You will also need `pgpassword` with the postgres password. All attributes pointing to Secret Keys have a default value, and in this example we are relying on the default values of `config.macaroonSecretKey` : `macaroon`, `config.registrationSharedSecretSecretKey` : `registrationSharedSecret`, `config.signingKeySecretKey` : `signingKey` and the `config.adminPasswordSecretKey` pointing to `adminPassword` in the secret key.
- `name: external-wellknowndelegation-secrets` with data keys `wellknown.tls` containing well known delegation private key and `wellknown.crt` containing well known delegation certificate.

Once the CRD and the Secrets deployed to the namespace, the Updater will be able to create all the resources handled by the Operator, which will then deploy the workloads on your kubernetes cluster.

Loading docker secrets into kubernetes in preparation of deployment

N.B. This guide assumes that you are using the `element-onprem` namespace for deploying Element. You can call it whatever you want and if it doesn't exist yet, you can create it with: `kubectl create ns element-onprem`.

Now we need to load secrets into kubernetes so that the deployment can access them. If you built your own CRD from scratch, you will need to follow our Element Deployment CRD documentation.

```
kubectl create secret -n element-onprem docker-registry ems-image-store --docker-server=registry.element.io --docker-username=<EMSusername> --docker-password=<EMStoken>
```

Checking deployment progress

To check on the progress of the deployment, you will first watch the logs of the updater:

```
kubectl logs -f -n element-updater element-updater-controller-manager-<rest of pod name>
```

You will have to tab complete to get the correct hash for the element-updater-controller-manager pod name.

Once the updater is no longer pushing out new logs, you can track progress with the operator or by watching pods come up in the `element-onprem` namespace.

Operator status:

```
kubectl logs -f -n element-operator element-operator element-operator-controller-manager-<rest of pod name>
```

Watching reconciliation move forward in the `element-onprem` namespace:

```
kubectl get elementdeployment -o yaml | grep dependentCRs -A20 -n element-onprem -w
```

Watching pods come up in the `element-onprem` namespace:

```
kubectl get pods -n element-onprem -w
```

Revision #25
Created 12 July 2023 17:47:38 by Karl Abbott
Updated 27 November 2024 12:07:50 by Kieran Mitchell Lane