HCLSoftware

Software Containerization Lesson 4

l.ziosi@vu.nl

Agenda

- Storage
  - Volumes
  - Persistent Volume
  - Persistent Volume Claim
  - Storage Classes and Dynamic Volume provisioning
- StatefulSet
- Configuration
  - ConfigMaps
  - Secrets
  - Example: deploying Postgresql to Kubernetes
  - X509 Certificates
  - TLS Secrets
  - Configuring certificate on Ingress
  - Cert-Manager

HCLSoftware

Kubernetes Storage

github.com/container-storage-interface/spec/blob/master/spec.md

Volume Lifecycle

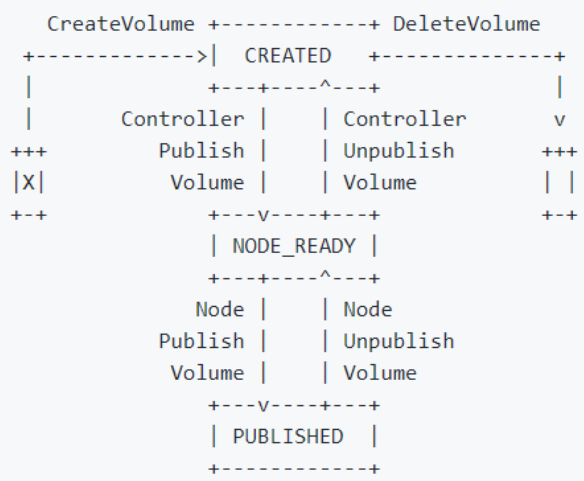


Figure 5: The lifecycle of a dynamically provisioned volume, from creation to destruction.

Storage in Kubernetes: volumes

In Kubernetes, pods host containers, that are based on container images which have a writable layer.

If a container crashes and is restarted, the data in the writable layer of the container is lost.

Kubernetes introduces volumes to provide containers inside a pod with sharable storage that survives crashes of the containers.

Volumes in Kubernetes are attached to the lifecycle of the pod, not of the containers inside the pod.

You create a volume as part of a pod, and mount it in every container (possibly, at different mount points).

The volume will exist as long as the pod exists.

What actually happens to the backing storage of the volume when the pod ceases to exist, depends on the type of volume.

Storage in Kubernetes: volumes

A volume is a directory, which is accessible to the containers in a pod.

A process in a container accesses a composite filesystem following these rules:

- at the root there is the filesystem of the container image
- volumes are mounted at specified paths within the image
- volumes can not be mounted onto other volumes or have hard links to other volumes
- each container in the pod must specify where to mount each volume

There are many types of volumes, depending on the type of environment:

- public cloud (example: gcePersistentDisk in Google Cloud Platform)
- on-premise installation (example: emptyDir, hostPath, nfs)

emptyDir volume

- An emptyDir volume is created when a Pod is assigned to a node.
- An emptyDir volume exists as long as the Pod is running on the node.
- An emptyDir volume is initially empty.
- All containers in the Pod can read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each container.
- When a Pod is removed from a node, the data in the emptyDir is deleted permanently.
- A container crashing does not remove a Pod from a node. The data in an emptyDir volume is safe across container crashes.
- emptyDir volumes are stored on the node storage, be it disk or SSD, or network storage.
- If you set the emptyDir.medium field to "Memory", Kubernetes mounts a tmpfs (RAM-backed filesystem). tmpfs is cleared on node reboot and any files you write count against your container's memory limit.

emptyDir example

- One volume of type emptyDir is added to a Pod
- The volume is mounted by two containers using the same mountPath
- Each container section needs to specify a volumeMounts map that contains an array of objects with a name and a mountPath.
- Each mountPath specifies the location inside the file system of the container where the volume can be access.
- Executing the following command creates the pod, the containers and the volume:
- `kubectl apply -f empty-dir.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: empty-dir
spec:
  containers:
    - name: alpine-a
      command: ['tail', '-f', '/dev/null']
      image: alpine
      imagePullPolicy: IfNotPresent
      volumeMounts:
        - name: cache
          mountPath: /cache
    - name: alpine-b
      command: ['tail', '-f', '/dev/null']
      image: alpine
      imagePullPolicy: IfNotPresent
      volumeMounts:
        - name: cache
          mountPath: /cache
  volumes:
    - name: cache
      emptyDir: {}
```

https://codeburst.io/kubernetes-storage-by-example-part-1-27f44ae8fb8b

emptyDir example continued

We can then demonstrate that if we write data to the /cache directory of the first container, it can be read by the second container.

In this example, emptyDir functions like a cache shared between the containers in the same pod.

```
lara@kube-master-gui:~/k8s_services$ kubectl exec empty-dir --container alpine-a -- sh -c "echo \"Hello World\" > /cache/hello.txt"
lara@kube-master-gui:~/k8s_services$ kubectl exec empty-dir --container alpine-b -- cat /cache/hello.txt
Hello World
lara@kube-master-gui:~/k8s_services$ kubectl exec empty-dir --container alpine-a -- sh -c "echo \"Hello Lara\" > /cache/hello.txt"
lara@kube-master-gui:~/k8s_services$ kubectl exec empty-dir --container alpine-b -- cat /cache/hello.txt
Hello Lara
```

Volume example: hostPath

A volume of type hostPath allows the containers to access a path in the filesystem of the node on which the pod runs.

Note that the files on that filesystem will be owned by user root and group root.

hostPath makes sense to get familiar with volumes when working with microk8s in a single node scenario, but only in rare cases it’s helpful in multiple node scenarios, because typically the scheduler is free to install pods on arbitrary nodes and therefore it’s unclear on which node the hostPath storage will be used.

Volume example: creating a hostPath volume in a Pod

- To specify a volume, you need to add the volumes map to the spec section.
- Each volume contains a name and a hostPath map that specifies the path and (optionally) the type.
- There are many possible values for type, including files, directories and sockets, as documented here.
- Each container section needs to specify a volumeMounts map that contains an array of objects with a name and a mountPath.
- Each mountPath specifies the location inside the file system of the container where the volume can be access.

```
apiVersion: v1
kind: Pod
metadata:
  name: host-path
spec:
  containers:
    - name: host-path-container
      command: ['tail', '-f', '/dev/null']
      image: alpine
      imagePullPolicy: IfNotPresent
      volumeMounts:
        - name: data
          mountPath: /data
  volumes:
  - name: data
    hostPath:
      path: /opt/data
```

Usage of hostPath

In the example below, you can see that the user on the host creates a file inside the directory mounted by the container in the pod. To do this, sudo is required as the directory is owned by root.

Then you can execute the ls command inside the container and notice hos the container see the file added on the host.

```
lara@kube-master-gui:~/k8s_services$ sudo touch /opt/data/hello.txt
lara@kube-master-gui:~/k8s_services$ ls -al /opt/data
total 8
drwxr-xr-x 2 root root 4096 jan 12 23:22 .
drwxr-xr-x 6 root root 4096 jan 12 23:20 ..
-rw-r--r-- 1 root root    0 jan 12 23:22 hello.txt
lara@kube-master-gui:~/k8s_services$ kubectl exec host-path -- ls /data
hello.txt
```

Volume example: nfs

- A volume of type nfs allows an existing NFS (Network File System) share to be mounted into a Pod.
- Kubernetes does not provide NFS, you must install and run your own NFS server and you must export the share before you can use it in a Kubernetes nfs volume.
- When a Pod is removed, the contents of an nfs volume used by the pod are preserved and the volume is just unmounted.
- An NFS volume can be pre-populated with data
- The data in an nfs volume data can be shared between pods.
- An nfs volume can be mounted by multiple writers simultaneously.

- https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-volumes-example-nfs-persistent-volume.html

Persistent Volume and Persistent Volume Claim

The volumes we have seen so far get created when the pod is created and get destroyed when the pod ceases to exist.

Persistent Volumes have a lifecycle that is decoupled from that of pods.

PVs can be created independently from pods, by a system administrator that is responsible for storage management.

Pods are created by developers, who can specify Persistent Volume Claims, that express the requirements of the application in terms of storage.

Kubernetes binds Persistent Volume Claims to Persistent Volumes.

Persistent Volume Claims may specify the storage requirements of the application in terms of:

- **Capacity** (units of measure are specified here)
- **AccessModes**:
  - ReadWriteOnce, ReadOnlyMany or ReadWriteMany

Kubernetes will try to find a Persistent Volume that matches the requirements of the Claim. It may find a PV that is larger than the PVC requires, in which case the extra space will remain unused. PV and PVC are a one to one mapping.

Access Modes

The access modes are:

- **ReadWriteOnce** -- the volume can be mounted as read-write by a single node
- **ReadOnlyMany** -- the volume can be mounted read-only by many nodes
- **ReadWriteMany** -- the volume can be mounted as read-write by many nodes

In the CLI, the access modes are abbreviated to:

- **RWO** - ReadWriteOnce
- **ROX** - ReadOnlyMany
- **RWX** – ReadWriteMany

NFS supports RWO, ROX, RWX, but a specific NFS PV might be exported on the server as read-only.

Each PV gets its own set of access modes describing that specific PV's capabilities.

Note that GCEPersistentDisk does not support RWX. For the complete list of what providers support see:

https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes

Static and Dynamic provisioning

PVs may be provisioned statically or dynamically.

Static provisioning

PVs are created in advance by an administrator, with all their details about storage specified. They are ready to be consumed by PVCs, but they may or may not fulfill the needs of the PVCs that will be created later by developers.

Dynamic provisioning (based on StorageClasses)

When none of the static PVs matches a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume that matches the requirements of the PVC.

To enable dynamic storage provisioning, the cluster administrator needs to enable the **DefaultStorageClass admission controller** on the API server.

The PVC must request a storage class and the administrator must have created and configured that class.

If the claims requests the class ""“, it means that it does not accept dynamic provisioning.

With dynamic provisioning, there is no problem of wasted storage due to PVs that are larger than the PVC.

Microk8s storage support

Microk8s has a storage add-on that **c**reates a **default storage class** which allocates storage from a host directory.

The name of this storage class is **microk8s-hostpath** as can be seen from the command:

```
lara@kube-master-gui:~/k8s_services$ kubectl get sc
NAME                          PROVISIONER            RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
microk8s-hostpath (default)   microk8s.io/hostpath   Delete          Immediate           false                  9d
```

```
lara@kube-master-gui:~/k8s_services$ kubectl get sc microk8s-hostpath -o yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"storage.k8s.io/v1","kind":"StorageClass","metadata":{"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}
,"name":"microk8s-hostpath"},"provisioner":"microk8s.io/hostpath"}
    storageclass.kubernetes.io/is-default-class: "true"
  creationTimestamp: "2021-01-03T14:04:04Z"
  managedFields:
  - apiVersion: storage.k8s.io/v1
    fieldsType: FieldsV1
    fieldsV1:
      f:metadata:
        f:annotations:
          .: {}
          f:kubectl.kubernetes.io/last-applied-configuration: {}
          f:storageclass.kubernetes.io/is-default-class: {}
      f:provisioner: {}
      f:reclaimPolicy: {}
      f:volumeBindingMode: {}
    manager: kubectl-client-side-apply
    operation: Update
    time: "2021-01-03T14:04:04Z"
  name: microk8s-hostpath
  resourceVersion: "228624"
  selfLink: /apis/storage.k8s.io/v1/storageclasses/microk8s-hostpath
  uid: bc386c9b-d97b-42dd-84b4-2ebd3c8a2522
provisioner: microk8s.io/hostpath
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

Persistent Volume example

PersistentVolume is an object in Kubernetes

accessMode ReadWriteOnce means it can be mounted read.write by a single node

hostPath refers to the location on the host where the PV is created

Capacity indicates the maximum size

storageClassName in this case is the default one of microk8s. If you don't specify it, it will be set to the default storage class if it exists. If you don't want to use a storage class, explicitly set it to an empty string "".

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: data
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
  hostPath:
    path: /opt/data
  storageClassName: microk8s-hostpath
```

```
lara@kube-master-gui:~/k8s_services$ kubectl apply -f pv.yaml
persistentvolume/data created
lara@kube-master-gui:~/k8s_services$ kubectl get pv data
NAME   CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS        REASON   AGE
data   1Gi        RWO            Retain           Available           microk8s-hostpath            10s
```

PersistentVolumeClaim example

When you create a PersistentVolumeClaim you need to specify the requirements (AccessMode, Capacity) and you may also add the desired StorageClass.

Note that the PVC got bound to the PV previously created, because it was available and it was matching the requirements.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: microk8s-hostpath
```

```
lara@kube-master-gui:~/k8s_services$ kubectl apply -f pv.yaml
persistentvolume/data created
lara@kube-master-gui:~/k8s_services$ kubectl apply -f pvc.yaml
persistentvolumeclaim/data-pvc created
lara@kube-master-gui:~/k8s_services$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                                 STORAGECLAS
S        REASON   AGE
pvc-33b7eeaa-f523-4445-9c1d-e404559e1308   20Gi       RWX            Delete           Bound    container-registry/registry-claim     microk8s-ho
stpath            3d6h
data                                       1Gi        RWO            Retain           Bound    default/data-pvc                      microk8s-ho
stpath            9s
lara@kube-master-gui:~/k8s_services$ kubectl get pvc
NAME       STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS        AGE
data-pvc   Bound    data     1Gi        RWO            microk8s-hostpath   10s
```

Creating PVC without creating PV (Dynamic provisioning)

If you create a new PVC definition file and just change the name of the PVC and apply again without first creating a new PV, the volume plugin will:

- Check that there is no available PV that satisfies the requirements of the claim.
- Create a new PV which matches the requirements of this new PVC. The name of the PV is automatically generated.
- Bind the PV and the PVC.

This happens only if there is a Default StorageClass defined in the cluster.

```
lara@kube-master-gui:~/k8s_services$ nano pvc2.yaml
lara@kube-master-gui:~/k8s_services$ kubectl apply -f pvc2.yaml
persistentvolumeclaim/data-pvc2 created
lara@kube-master-gui:~/k8s_services$ kubectl get pvc data-pvc2
NAME        STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS        AGE
data-pvc2   Bound    pvc-77045e48-f3d3-4c51-a014-f31c26c93f07   1Gi        RWO            microk8s-hostpath   13s
lara@kube-master-gui:~/k8s_services$ kubectl get pv pvc-77045e48-f3d3-4c51-a014-f31c26c93f07
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM               STORAGECLASS        REASON
  AGE
pvc-77045e48-f3d3-4c51-a014-f31c26c93f07   1Gi        RWO            Delete           Bound    default/data-pvc2   microk8s-hostpath
  30s
```

Usage of PersistentVolumeClaim in a pod

To use the PVC in a pod, you must add a volume declaration that specifies a claim name matching an existing PVC.

Inside each container, you can then specify a volume mount that refers to the name of the volume and the mount point inside the container.

Then once the pod is created, the container can access the contents of the volume as shown below.

Since the PVC had access RWO, the container can write to the volume.

```
lara@kube-master-gui:~/k8s_services$ ls -al /opt/data
total 8
drwxr-xr-x 2 root root 4096 jan 13 21:19 .
drwxr-xr-x 6 root root 4096 jan 12 23:20 ..
lara@kube-master-gui:~/k8s_services$ kubectl exec pvc-client-pod -- touch /data/hello.txt
lara@kube-master-gui:~/k8s_services$ kubectl exec pvc-client-pod -- ls -l /data
total 0
-rw-r--r--    1 root     root             0 Jan 13 20:20 hello.txt
lara@kube-master-gui:~/k8s_services$ ls -al /opt/data
total 8
drwxr-xr-x 2 root root 4096 jan 13 21:20 .
drwxr-xr-x 6 root root 4096 jan 12 23:20 ..
-rw-r--r-- 1 root root    0 jan 13 21:20 hello.txt
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pvc-client-pod
spec:
  containers:
    - name: pvc-client-container
      command: ['tail', '-f', '/dev/null']
      image: alpine
      imagePullPolicy: IfNotPresent
      volumeMounts:
        - name: data
          mountPath: /data
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: data-pvc
```

Reclaiming

The Reclaim Policy on PVs describes what happens to the storage and the data after you delete the PersistentVolumeClaim.

There are 3 policies:

- **Retain**: When the PVC is deleted, the PV is not deleted. The Admin can then delete the PV but this does not delete the storage and the data which must then be deleted manually.
- **Delete**: When the PVC is deleted ,the PV is deleted and the underlying storage as well. Volumes that were dynamically provisioned inherit the reclaim policy of their StorageClass, which defaults to Delete.
- Recycle (deprecated)

```
lara@kube-master-gui:~/k8s_services$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                                STORAGECLAS
S        REASON   AGE
pvc-33b7eeaa-f523-4445-9c1d-e404559e1308   20Gi       RWX            Delete           Bound    container-registry/registry-claim    microk8s-ho
stpath             3d6h
data                                       1Gi        RWO            Retain           Bound    default/data-pvc                     microk8s-ho
stpath             30m
pvc-77045e48-f3d3-4c51-a014-f31c26c93f07   1Gi        RWO            Delete           Bound    default/data-pvc2                    microk8s-ho
stpath             27m
```

Reclaiming example

Observe the different results of deleting data-pvc2 which was provisioned dynamically (Reclaim policy = Delete)

versus deleting data-pvc that was provisioned statically (Reclaim policy = Retain).

The volume data bound to data-pvc is not deleted and its status is set to Released.

```
lara@kube-master-gui:~/k8s_services$ kubectl get pvc
NAME        STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS        AGE
data-pvc    Bound    data                                       1Gi        RWO            microk8s-hostpath   34m
data-pvc2   Bound    pvc-77045e48-f3d3-4c51-a014-f31c26c93f07   1Gi        RWO            microk8s-hostpath   31m
lara@kube-master-gui:~/k8s_services$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                              STORAGECLAS
S        REASON   AGE
pvc-33b7eeaa-f523-4445-9c1d-e404559e1308   20Gi       RWX            Delete           Bound    container-registry/registry-claim   microk8s-ho
stpath            3d6h
data                                       1Gi        RWO            Retain           Bound    default/data-pvc                    microk8s-ho
stpath            35m
pvc-77045e48-f3d3-4c51-a014-f31c26c93f07   1Gi        RWO            Delete           Bound    default/data-pvc2                   microk8s-ho
stpath            32m
lara@kube-master-gui:~/k8s_services$ kubectl delete pvc data-pvc2
persistentvolumeclaim "data-pvc2" deleted
lara@kube-master-gui:~/k8s_services$ kubectl delete pvc data-pvc
persistentvolumeclaim "data-pvc" deleted
lara@kube-master-gui:~/k8s_services$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM                              STORAGECL
ASS        REASON   AGE
pvc-33b7eeaa-f523-4445-9c1d-e404559e1308   20Gi       RWX            Delete           Bound      container-registry/registry-claim   microk8s-
hostpath           3d6h
data                                       1Gi        RWO            Retain           Released   default/data-pvc                    microk8s-
hostpath           35m
```

Container Storage Interface

The Container Storage Interface specification is defined at:
https://github.com/container-storage-interface/spec/blob/master/spec.md

It establishes the interfaces and architectural requirements that enable storage vendors (SP) to develop storage plugins that can work in a number of container orchestration (CO) systems.

The specification defines APIs (Remote Procedure Calls) that enable:

- Dynamic provisioning and deprovisioning of a volume.
- Attaching or detaching a volume from a node.
- Mounting/unmounting a volume from a node.
- Consumption of both block and mountable volumes.
- Local storage providers (e.g., device mapper, lvm).
- Creating and deleting a snapshot (source of the snapshot is a volume).
- Provisioning a new volume from a snapshot (reverting snapshot, where data in the original volume is erased and replaced with data in the snapshot, is out of scope).

HCLSoftware

StatefulSet

StatefulSet

StatefulSet is the workload API object that you can use instead of Deployment if you need to manage a stateful application (such as a relational or no-sql database).

Like a Deployment, a StatefulSet manages a set of pods that are created from the same template, and maintains the desired number of replicas of these pods.

Contrary to a Deployment, a StatefulSet ensures that you have a unique and easy way of identifying each pod, and that the pods are created in a specific order.

Although the pods are created from the same template specification, they are not interchangeable.

This is useful for example if you need to deploy a highly available configuration of a database with one master and two slaves. If one of the pods crashes and needs to be recreated, it is important that Kubernetes knows which specific one it needs to recreate (master, slave-1 or slave-2).

If the application needs storage volumes, using StatefulSets makes it easier to couple the existing volumes to the new Pods that replace any that have failed.

Headless Service definition

In order to create a Statefulset, you need to create first a Headless Service.

A HeadlessService is a Service object in which you set the ClusterIP to None.

In headless Service, a cluster IP is not allocated.

kube-proxy does not handle these Services.

There is no load balancing or proxying done by Kubernetes.

Note that the selector must match some label defined on the pods of the StatefulSet.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  labels:
    app: nginx-app
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx-app
```

StatefulSet definition

The StatefulSet definition contains a reference to the headless Service previously defined (nginx-service).

It specifies replicas, like a Deployment or ReplicaSet would.

It specifies a persistent volume claim template instead of a persistent volume claim.

This is because each of the pods that will be created will get its own persistent volume claim and its own persistent volume.

If you are deploying a highly available database, each of the database servers (pods) would need to have its own storage besides a way to replicate the data from master to slave (through log replication and similar technologies).

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx-service"
  replicas: 2
  selector:
    matchLabels:
      app: nginx-app
  template:
    metadata:
      labels:
        app: nginx-app
    spec:
      containers:
      - name: nginx-container
        image: nginx
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Observe the ordered creation of the pods

To observe the ordered creation of the pods, you can use the watch feature of kubectl (-w), that repeats the command at time intervals so you can see how the system evolves over time:

```
lara@kube-master-gui:~/k8s_services$ kubectl get pods -w -l app=nginx-app
NAME    READY   STATUS    RESTARTS   AGE
web-0   0/1     Pending   0          0s
web-0   0/1     Pending   0          0s
web-0   0/1     Pending   0          8s
web-0   0/1     ContainerCreating   0          8s
web-0   0/1     ContainerCreating   0          9s
web-0   1/1     Running             0          11s
web-1   0/1     Pending             0          0s
web-1   0/1     Pending             0          0s
web-1   0/1     Pending             0          8s
web-1   0/1     ContainerCreating   0          8s
web-1   0/1     ContainerCreating   0          9s
web-1   1/1     Running             0          11s
```

The two pods have ordinal numbers appended to the name.
The first pod is completely initialized (Running state) before the second pod enters the Pending state.

In a second terminal you can launch the creation of the Service and the StatefulSet and watch what happens in the first terminal.

```
lara@kube-master-gui:~/k8s_services$ kubectl apply -f stateful.yaml
service/nginx-service created
statefulset.apps/web created
```

Observe the creation of the PVCs and PVs

Note that the StatefulSet declared a persistent volume claim template.

This caused the generation of a persistent volume claim for each pod.

In turn, each persistent volume claim was bound to a dynamically generated persistent volume.

```
lara@kube-master-gui:~/k8s_services$ kubectl get pvc
NAME        STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS        AGE
www-web-0   Bound    pvc-df091459-a6a1-495e-9b7f-1af5a2649f3e   1Gi        RWO            microk8s-hostpath   5m20s
www-web-1   Bound    pvc-1dafc864-e9f3-401b-b7b7-5caae503e7e0   1Gi        RWO            microk8s-hostpath   5m9s
lara@kube-master-gui:~/k8s_services$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                                  STORAGECLASS        REASON   AGE
pvc-33b7eeaa-f523-4445-9c1d-e404559e1308   20Gi       RWX            Delete           Bound    container-registry/registry-claim      microk8s-hostpath            3d7h
pvc-df091459-a6a1-495e-9b7f-1af5a2649f3e   1Gi        RWO            Delete           Bound    default/www-web-0                      microk8s-hostpath            5m18s
pvc-1dafc864-e9f3-401b-b7b7-5caae503e7e0   1Gi        RWO            Delete           Bound    default/www-web-1                      microk8s-hostpath            5m7s
```

For more details of the StatefulSet, see: https://kubernetes.io/docs/tutorials/stateful-application/basic-stateful-set/

HCLSoftware

Configuration:

ConfigMaps

ConfigMaps

12 factor apps is a framework of best practices for designing cloud native applications: https://12factor.net/
One key aspect is Config: store config in the environment

This indicates the need for separating the application code (which should be environment-independent) from configurations that depend on the environment (dev, test, staging, prod, but also personal laptop vs public cloud).

Kubernetes implements this best practice with **ConfigMap** for data that is not secret and **Secret** for passwords, tokens, certificates etc.

Examples of data to store in ConfigMap are hostnames or URLs of external services like databases.

A ConfigMap is an API object used to store non-confidential data in key-value pairs.

Pods can get data from ConfigMaps in the following 4 ways:

- Environment variables declared in the spec of the container inside the pod
- Command-line arguments passed to a container command inside the pod
- Read-only configuration files in a volume that the pod can read
- Container code that calls the Kubernetes REST API to get the data from the ConfigMap object

ConfigMap object with simple key value pairs

The simplest ConfigMap object contains individual key value pairs.

A pod can get these values injected as Environment variables by declaring an env key and listing the names of the environment variable followed by references to the ConfigMap name and key there contained.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config-map
data:
  # property-like keys
  # each key maps to a simple value
  database_host1: "mydb1.mydomain.com"
  database_host2: "mydb2.mydomain.com"
```

```
apiVersion: v1
kind: Pod
metadata:
  name: db-config-map-client-pod
spec:
  containers:
    - name: db-config-map-client-container
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variables
        - name: DATABASE_HOST1
          valueFrom:
            configMapKeyRef:
              name: db-config-map
              key: database_host1
        - name: DATABASE_HOST2
          valueFrom:
            configMapKeyRef:
              name: db-config-map
              key: database_host2
```

Creation of the ConfigMap object

The ConfigMap object is created by applying the configuration file.
You can then get the details of the ConfigMap using describe which prints out the Data section

```
lara@kube-master-gui:~/k8s_services$ kubectl apply -f config-map1.yaml
configmap/db-config-map created
lara@kube-master-gui:~/k8s_services$ kubectl get configmap
NAME               DATA   AGE
kube-root-ca.crt   1      12d
db-config-map      2      16s
lara@kube-master-gui:~/k8s_services$ kubectl describe configmap db-config-map
Name:         db-config-map
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
database_host1:
----
mydb1.mydomain.com
database_host2:
----
mydb2.mydomain.com
Events:  <none>
```

How the pod accesses environment variables from ConfigMap

```
lara@kube-master-gui:~/k8s_services$ kubectl describe pod db-config-map-client-pod
Name:         db-config-map-client-pod
Namespace:    default
Priority:     0
Node:         kube-master-gui/10.0.2.15
Start Time:   Wed, 13 Jan 2021 14:16:27 +0100
Labels:       <none>
Annotations:  cni.projectcalico.org/podIP: 10.1.96.181/32
              cni.projectcalico.org/podIPs: 10.1.96.181/32
Status:       Running
IP:           10.1.96.181
IPs:
  IP:  10.1.96.181
Containers:
  db-config-map-client-container:
    Container ID:  containerd://bc43cdadf2b6e9d0706d294a95ca5e7475053ff3c371f4a6eef6c27c99052244
    Image:         alpine
    Image ID:      docker.io/library/alpine@sha256:3c7497bf0c7af93428242d6176e8f7905f2201d8fc5861f45be7a346b5f23436
    Port:          <none>
    Host Port:     <none>
    Command:
      sleep
      3600
    State:          Running
      Started:      Wed, 13 Jan 2021 14:16:32 +0100
    Ready:          True
    Restart Count:  0
    Environment:
      DATABASE_HOST1:  <set to the key 'database_host1' of config map 'db-config-map'>  Optional: false
      DATABASE_HOST2:  <set to the key 'database_host2' of config map 'db-config-map'>  Optional: false
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-tnd79 (ro)
```

```
lara@kube-master-gui:~/k8s_services$ kubectl exec -it db-config-map-client-pod -- ash
/ # printenv |grep DATABASE
DATABASE_HOST1=mydb1.mydomain.com
DATABASE_HOST2=mydb2.mydomain.com
/ #
```

Creating a ConfigMap from literals on the command line

You can create a ConfigMap from literals using the command line as follows:

```
lara@kube-master-gui:~/k8s_services$ kubectl create configmap special-config --from-literal=mykey.1=myvalue.1 --from-literal=mykey.2=myvalue.2
configmap/special-config created
lara@kube-master-gui:~/k8s_services$ kubectl describe configmap special-config
Name:         special-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
mykey.1:
----
myvalue.1
mykey.2:
----
myvalue.2
Events:  <none>
```

Creating a ConfigMap from files

```
lara@kube-master-gui:~/k8s_services$ cat - > my-conf.properties
property1=value1
property2=value2
property3=value3
```

```
lara@kube-master-gui:~/k8s_services$ kubectl create configmap my-conf --from-file=my-conf.properties
configmap/my-conf created
```

```
lara@kube-master-gui:~/k8s_services$ kubectl describe configmap my-conf
Name:         my-conf
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
my-conf.properties:
----
property1=value1
property2=value2
property3=value3

Events:  <none>
```

```
lara@kube-master-gui:~/k8s_services$ kubectl get configmap my-conf -o yaml
apiVersion: v1
data:
  my-conf.properties: |
    property1=value1
    property2=value2
    property3=value3
kind: ConfigMap
metadata:
  creationTimestamp: "2021-01-13T21:29:31Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:my-conf.properties: {}
    manager: kubectl-create
    operation: Update
    time: "2021-01-13T21:29:31Z"
  name: my-conf
  namespace: default
  resourceVersion: "1358076"
  selfLink: /api/v1/namespaces/default/configmaps/my-conf
  uid: 978b1450-e9b3-4440-a76b-b8ddef134f6f
```

Secret

Data in a Secret object is not encrypted, but it is encoded using base 64 encoding (most of the time, unencoded strings are allowed).

This means that it can also be decoded without any need to supply any key. While this is practical, it also means that additional security measures must be taken to protect the data.

Secrets can have many types.

The Opaque type is the default if no other type is specified.

Builtin Type	Usage
`Opaque`	arbitrary user-defined data
`kubernetes.io/service-account-token`	service account token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file
`kubernetes.io/basic-auth`	credentials for basic authentication
`kubernetes.io/ssh-auth`	credentials for SSH authentication
`kubernetes.io/tls`	data for a TLS client or server
`bootstrap.kubernetes.io/token`	bootstrap token data

https://kubernetes.io/docs/concepts/configuration/secret/

Creating a Secret from a file

To create an Opaque Secret from a file, you need to follow the template shown and supply each secret value in the **data** section as a key-value pair, in base64 encoded format.

There is also a **stringData** field, where you can provide data as unencoded strings.

The name of a Secret object must be a valid DNS subdomain name.

The values in the data section need to be base-64 encoded.

On Linux you can encode by piping strings through the base64 command:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: TXlQYXNzd29yZA==
```

```
lara@kube-master-gui:~/k8s_services$ echo -n 'admin' | base64
YWRtaW4=
lara@kube-master-gui:~/k8s_services$ echo -n 'MyPassword' | base64
TXlQYXNzd29yZA==
```

You can then decode by adding –d parameter to the base64 command:

```
echo 'TXlQYXNzd29yZA=='|base64 -d
MyPassword
```

Creating and describing Secrets

You create a Secret with the usual command.
When you get the Secret you can see what type it is and how many data entries it contains.
When you describe the Secret you can see the names of the keys and their length, but not the values (contrary to ConfigMap).

```
lara@kube-master-gui:~/k8s_services$ kubectl apply -f secret1.yaml
secret/mysecret created
lara@kube-master-gui:~/k8s_services$ kubectl get secret mysecret
NAME       TYPE     DATA   AGE
mysecret   Opaque   2      12s
lara@kube-master-gui:~/k8s_services$ kubectl describe secret mysecret
Name:         mysecret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  10 bytes
username:  5 bytes
```

Getting the yaml format shows the data in base64 encoded format.

```
lara@kube-master-gui:~/k8s_services$ kubectl get secret mysecret -o yaml
apiVersion: v1
data:
  password: TXlQYXNzd29yZA==
  username: YWRtaW4=
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"password":"TXlQYXNzd29yZA==","username":"YWRt
aW4="},"kind":"Secret","metadata":{"annotations":{},"name":"mysecret","namespac
e":"default"},"type":"Opaque"}
  creationTimestamp: "2021-01-13T15:11:04Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:password: {}
        f:username: {}
      f:metadata:
        f:annotations:
          .: {}
          f:kubectl.kubernetes.io/last-applied-configuration: {}
      f:type: {}
    manager: kubectl-client-side-apply
    operation: Update
    time: "2021-01-13T15:11:04Z"
  name: mysecret
  namespace: default
  resourceVersion: "1304036"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: 740f4629-0ddf-4915-86d5-41a3f3ec48b1
type: Opaque
```

Consuming Secrets from Pods as environment variables

A Pod can add to the container specification environment variables declarations that refer to Secrets by specifying the Secret name and the desired key from the Secret.

Note that even if the data in the Secret is base64 encoded, once the environment variable is injected inside the container its value is the decoded string (plain text).

```
lara@kube-master-gui:~/k8s_services$ kubectl apply -f secret1-user.yaml
pod/secret-env-pod created
lara@kube-master-gui:~/k8s_services$ kubectl exec -it secret-env-pod -- ash
/ # printenv |grep SECRET
SECRET_PASSWORD=MyPassword
SECRET_USERNAME=admin
/ #
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: alpine
    command: ["sleep", "3600"]
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

Example: Deploy Postgresql database: create ConfigMap

Using the Docker image: https://hub.docker.com/_/postgres

Create the file postgres-config.yaml

This file defines two variables, for the database name and database user.

These variables are not secret so they can reside in a ConfigMap.

Apply the configuration with:

```
kubectl apply -f postgres-config.yaml
```

Verify the configuration with:

```
kubectl get configmap postgres-config
```

```
kubectl describe configmap postgres-config
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgres-config
  labels:
    app: postgres
data:
  POSTGRES_DB: postgresdb
  POSTGRES_USER: postgresadmin
```

https://severalnines.com/database-blog/using-kubernetes-deploy-postgresql

Example: Deploy Postgresql database: create Secret

Create a secret to store the database password:

Base-64 encode the password:

echo admin123 | base64

YWRtaW4xMjMK

Create the file postgres-secret.yaml with the highlighted code.

```
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secret
type: Opaque
data:
  POSTGRES_PASSWORD: YWRtaW4xMjMK
```

Apply the Secret with the command:

```
kubectl apply -f postgres-secret.yaml
```

Verify with the commands:

```
kubectl get secret postgres-secret
kubectl describe secret postgres-secret
```

Example: Deploy Postgresql database: create PersistentVolume

Make a directory at: /opt/postgres/data with:

```
sudo mkdir -p /opt/postgre/data
```

Create the file postgres-storage.yaml

Apply the storage with the command:

```
kubectl apply -f postgres-storage.yaml
```

Verify with the commands:

```
kubectl get pv
kubectl get pvc
```

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: postgres-pv-volume
  labels:
    type: local
    app: postgres
spec:
  storageClassName: microk8s-hostpath
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/opt/postgres/data"
---
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: postgres-pv-claim
  labels:
    app: postgres
spec:
  storageClassName: microk8s-hostpath
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

Example: Deploy Postgresql database: create Deployment

Create the file postgres-deployment.yaml

Notice the environment variables from the configMapRef

Notice the password that comes from the secretKeyRef

The volume is mounted at a local path and it refers to a PersistentVolumeClaim

Apply the deployment with the command:

```
kubectl apply -f postgres-deployment.yaml
```

Verify with the commands:

```
kubectl get deployments
```

```
kubectl get pods -l app=postgres
```

```
apiVersion: app/v1
kind: Deployment
metadata:
  name: postgres-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
```

```
spec:
     containers:
       - name: postgres-container
         image: postgres:13.1-alpine
         imagePullPolicy: "IfNotPresent"
         ports:
           - containerPort: 5432
         envFrom:
           - configMapRef:
               name: postgres-config
         env:
           - name: POSTGRES_PASSWORD
             valueFrom:
               secretKeyRef:
                 name: postgres-secret
                 key: POSTGRES_PASSWORD
         volumeMounts:
           - mountPath: /var/lib/postgresql/data
             name: postgredb
     volumes:
       - name: postgredb
         persistentVolumeClaim:
           claimName: postgres-pv-claim
```

Example: Deploy Postgresql database: create NodePort service

Create the service file postgres-service.yaml:

Apply the service with the command:

```
kubectl apply -f postgres-service.yaml
```

Verify with the command:

```
kubectl get svc
```

Note: for security reasons, it is generally recommended to create just a ClusterIP for the database, but for testing, a NodePort helps you to connect from outside of the cluster which may help.

```
apiVersion: v1
kind: Service
metadata:
  name: postgres-service
  labels:
    app: postgres
spec:
  type: NodePort
  ports:
   - port: 5432
     nodePort: 30001
  selector:
   app: postgres
```

Example: test connecting to the database from the host VM

Ensure that the Postgres client is installed on the host, with:

```
sudo apt install postgresql-client
```

```
lara.ziosi@chatbot-test:~/github/hclswsupport-chatbot-data-import/k8s$ sudo apt install postgresql-client
[sudo] password for lara.ziosi:
Reading package lists... Done
Building dependency tree
Reading state information... Done
postgresql-client is already the newest version (10+190ubuntu0.1).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

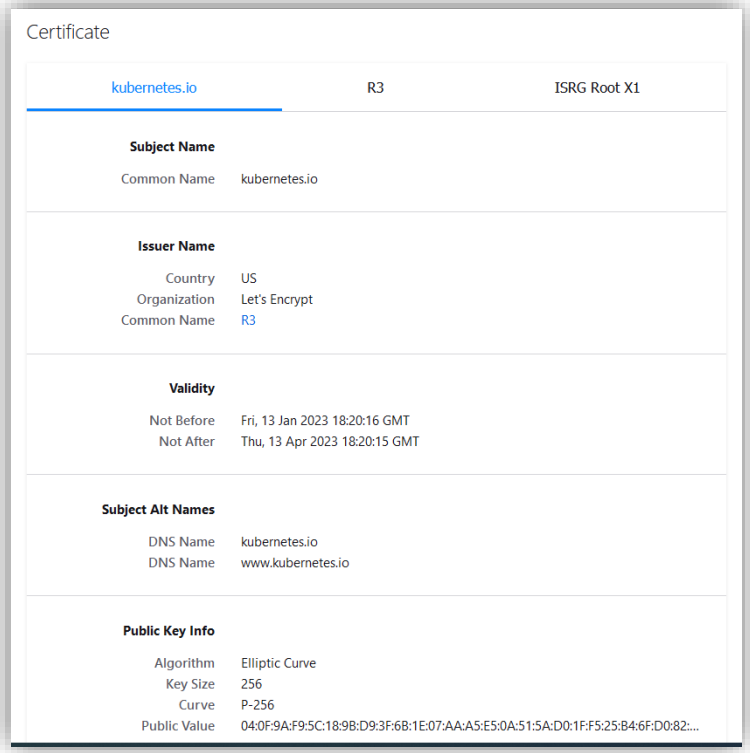
Connect to the database server with the client on the NodePort 30001, list the databases and exit:

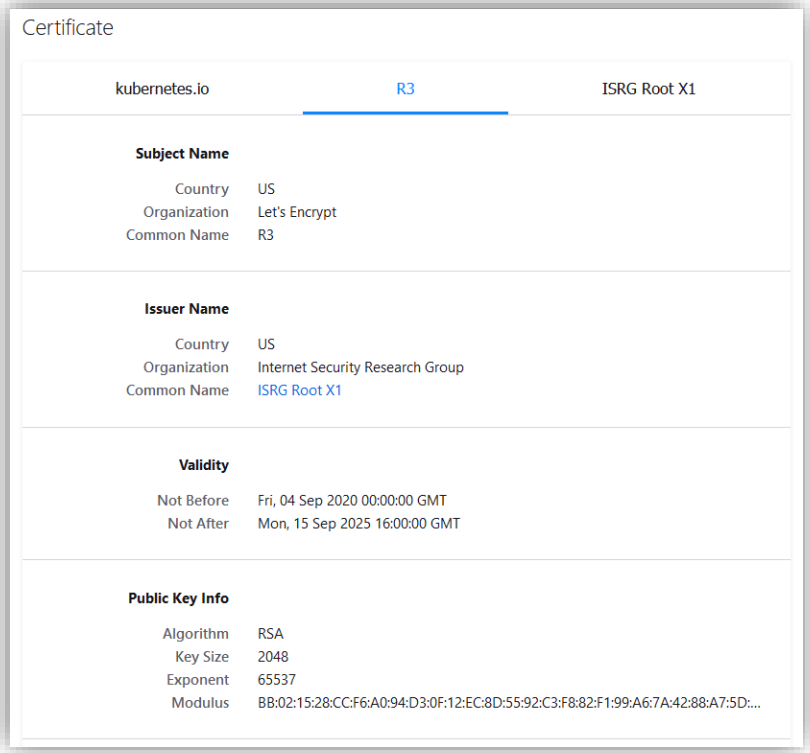
```
psql –h localhost –U postgresqladmin –password –p 30001 postgresdb
```

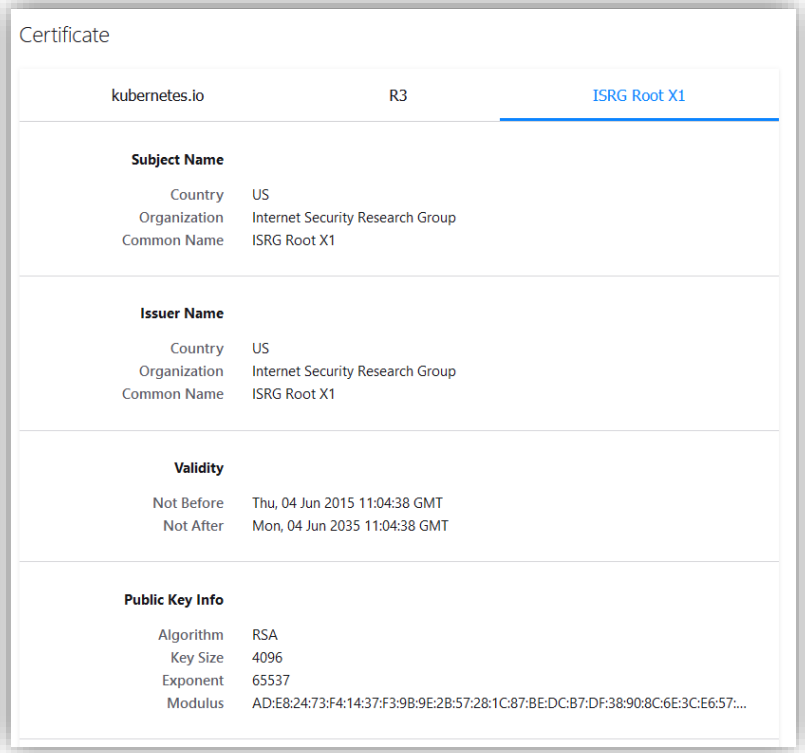
```
lara.ziosi@chatbot-test:~/github/hclswsupport-chatbot-data-import/k8s$ psql -h localhost -U postgresadmin --password -p 30001 postgresdb
Password for user postgresadmin:
psql (10.15 (Ubuntu 10.15-0ubuntu0.18.04.1), server 13.1)
WARNING: psql major version 10, server major version 13.
         Some psql features might not work.
Type "help" for help.

postgresdb=# \l
                                  List of databases
    Name    |     Owner     | Encoding |  Collate   |   Ctype    |       Access privileges
------------+---------------+----------+------------+------------+-------------------------------
 postgres   | postgresadmin | UTF8     | en_US.utf8 | en_US.utf8 |
 postgresdb | postgresadmin | UTF8     | en_US.utf8 | en_US.utf8 |
 template0  | postgresadmin | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgresadmin              +
            |               |          |            |            | postgresadmin=CTc/postgresadmin
 template1  | postgresadmin | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgresadmin              +
            |               |          |            |            | postgresadmin=CTc/postgresadmin
(4 rows)

postgresdb=# \q
```

Note that pods in the same cluster would be able to connect to the port which is 5432.

X509 certificate – intermediate CA – root CA







Web sites that use TLS are secured by X509 certificates (public and private key pairs).
Typically, certificates are signed by an **issuer** which is an Intermediate Certificate Authority (ICA)
The ICA is signed by a Certificate Authority (CA). Let's Encrypt is a CA that issues certificates free of charge.
The Subject (Alt) name needs to match the DNS name of the site (there are also wild card certificates).

https://en.wikipedia.org/wiki/X.509

TLS Secrets

When you need to configure applications with certificates to encrypt connections (like for terminating TLS on Ingress), you have the problem of keeping the certificate private keys secure.

These private keys can be stored in TLS Secrets together with the certificate itself (which is public).

The buil-tin Secret type **kubernetes.io/tls** is used for storing a certificate and its associated private key

The keys **tls.key** and **tls.crt** key must be provided in the data (or stringData) field of the Secret configuration, although the API server doesn't actually validate the values for each key.

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-tls
type: kubernetes.io/tls
data:
  # the data is abbreviated in this example
  tls.crt: |
        MIIC2DCCAcCgAwIBAgIBATANBgkqh ...
  tls.key: |
        MIIEpgIBAAKCAQEA7yn3bRHQ5FHMQ
...
```

How to create a TLS secret

The easiest way to create a TLS secret is from the command line, by supplying the files that contain the private key and the public key certificate. These files must be provided in PEM format. You can create the files with:

```
openssl req -x509 -newkey rsa:4090 -keyout key.pem -out cert.pem -days 365 -nodes
```

Once you have the two files `cert.pem` and `key.pem`, you can create the TLS Secret in Kubernetes with:

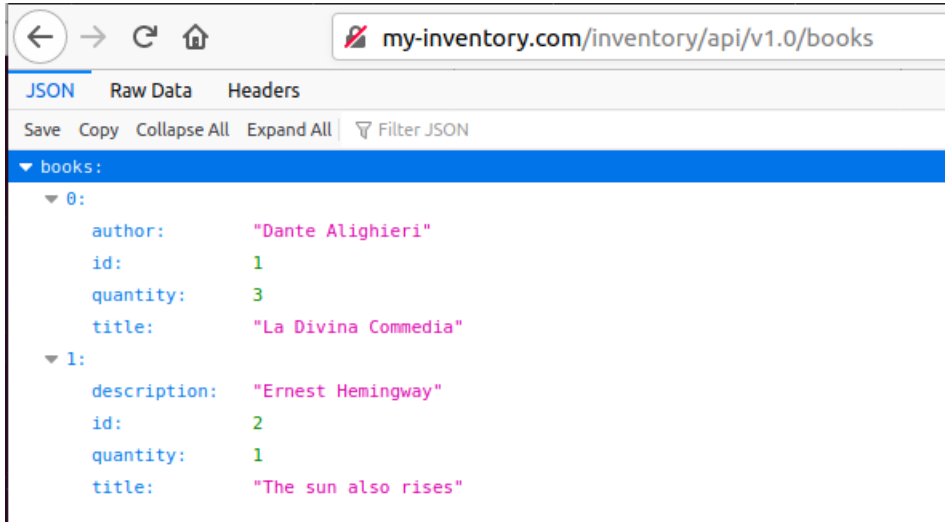
```
kubectl create secret tls my-tls-secret \
  --cert=cert.pem\
  --key=key.pem
```

```
lara@kube-master-gui:~/k8s_services$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes
Generating a RSA private key
.....................................................................................................................++++
.................................+++
writing new private key to 'key.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:NL
State or Province Name (full name) [Some-State]:Neteherlands
Locality Name (eg, city) []:Amsterdam
Organization Name (eg, company) [Internet Widgits Pty Ltd]:VU
Organizational Unit Name (eg, section) []:CS
Common Name (e.g. server FQDN or YOUR name) []:kube-master-gui
Email Address []:l.ziosi@vu.nl
lara@kube-master-gui:~/k8s_services$ kubectl create secret tls my-tls-secret \
>   --cert=cert.pem\
>   --key=key.pem
secret/my-tls-secret created
```

Using a TLS Secret to secure Ingress

https://kubernetes.github.io/ingress-nginx/user-guide/tls/
https://discuss.kubernetes.io/t/add-on-ingress/11259/2

In Lesson 3 we enabled Ingress just using the http protocol:



In the next slide we shall see how to enable Ingress for https with a default certificate valid for the entire cluster.

Note that it's not a good idea to just use a self-signed certificate. Even if you cannot buy a certificate from a CA or generate one for free with Let's Encrypt, you can at least create your own CA and sign the certificate with it:
https://www.baeldung.com/openssl-self-signed-cert

Disable and re-enable Ingress with default certificate

First disable ingress so it can be reconfigured:

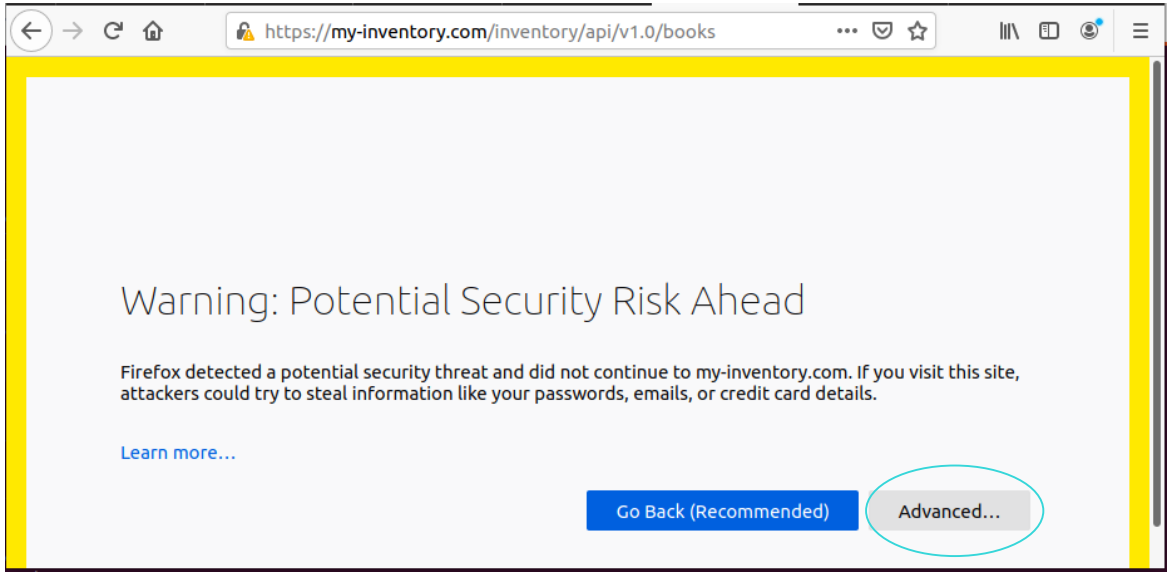
```
microk8s disable ingress
```

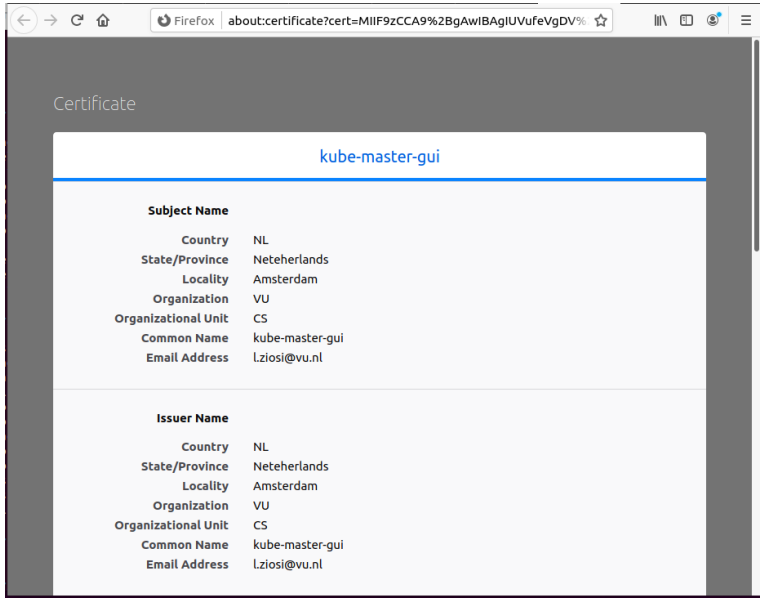
```
lara@kube-master-gui:~/k8s_services$ microk8s disable ingress
Disabling Ingress
ingressclass.networking.k8s.io "public" deleted
namespace "ingress" deleted
serviceaccount "nginx-ingress-microk8s-serviceaccount" deleted
clusterrole.rbac.authorization.k8s.io "nginx-ingress-microk8s-clusterrole" deleted
role.rbac.authorization.k8s.io "nginx-ingress-microk8s-role" deleted
clusterrolebinding.rbac.authorization.k8s.io "nginx-ingress-microk8s" deleted
rolebinding.rbac.authorization.k8s.io "nginx-ingress-microk8s" deleted
configmap "nginx-load-balancer-microk8s-conf" deleted
configmap "nginx-ingress-tcp-microk8s-conf" deleted
configmap "nginx-ingress-udp-microk8s-conf" deleted
daemonset.apps "nginx-ingress-microk8s-controller" deleted
Ingress is disabled
```

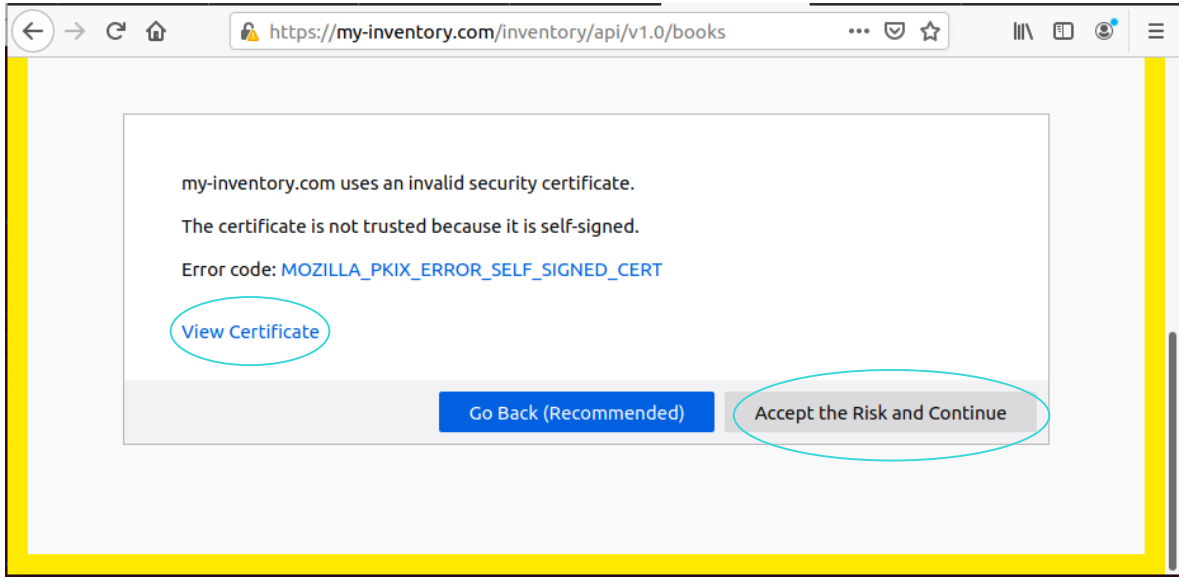
```
microk8s enable ingress:default-ssl-certificate=<namespace_name>/<secret_name>
```

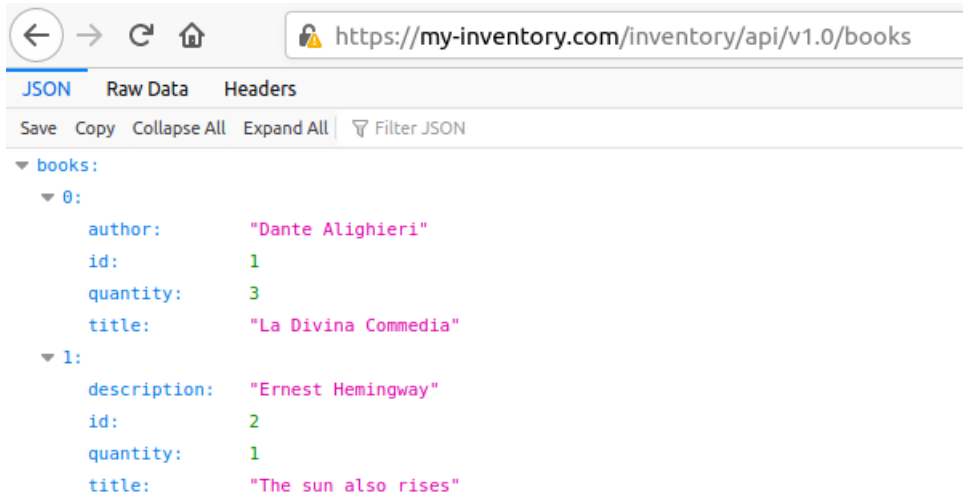
```
lara@kube-master-gui:~/k8s_services$ microk8s enable ingress:default-ssl-certificate=default/my-tls-secret
Enabling Ingress
Setting default/my-tls-secret as default ingress certificate
ingressclass.networking.k8s.io/public created
namespace/ingress created
serviceaccount/nginx-ingress-microk8s-serviceaccount created
clusterrole.rbac.authorization.k8s.io/nginx-ingress-microk8s-clusterrole created
role.rbac.authorization.k8s.io/nginx-ingress-microk8s-role created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress-microk8s created
rolebinding.rbac.authorization.k8s.io/nginx-ingress-microk8s created
configmap/nginx-load-balancer-microk8s-conf created
configmap/nginx-ingress-tcp-microk8s-conf created
configmap/nginx-ingress-udp-microk8s-conf created
daemonset.apps/nginx-ingress-microk8s-controller created
Ingress is enabled
```

Accessing sites served by Ingress over https with self-signed certificate









cert-manager

cert-manager is an external project that:

- adds certificates and certificate issuers as resource types in Kubernetes clusters
- simplifies the process of obtaining, renewing and using certificates.
- can issue certificates from – among others - Let's Encrypt, HashiCorp Vault and private PKI.
- verifies certificate validity and renews them before they expire

You can install cert-manager with:

```
kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.11.0/cert-manager.yaml
```

You must uninstall it with the same yaml file:

```
kubectl delete -f https://github.com/cert-manager/cert-manager/releases/download/v1.11.0/cert-manager.yaml
```

On Micro8s, you can enable it with:

```
microk8s enable cert-manager
```

https://cert-manager.io/docs/

Examine resources created by cert-manager

```
kubectl get all –n cert-manager
```

```
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get all -n cert-manager
NAME                                           READY   STATUS    RESTARTS   AGE
pod/cert-manager-cainjector-89769b4cb-xnfkl    1/1     Running   0          4m58s
pod/cert-manager-webhook-76bd54d4f-pnnmv       1/1     Running   0          4m58s
pod/cert-manager-69c6cb69f9-m7pn5              1/1     Running   0          4m58s

NAME                           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/cert-manager           ClusterIP   10.152.183.89    <none>        9402/TCP   4m58s
service/cert-manager-webhook   ClusterIP   10.152.183.166   <none>        443/TCP    4m58s

NAME                                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/cert-manager-cainjector   1/1     1            1           4m58s
deployment.apps/cert-manager-webhook      1/1     1            1           4m58s
deployment.apps/cert-manager              1/1     1            1           4m58s

NAME                                                 DESIRED   CURRENT   READY   AGE
replicaset.apps/cert-manager-cainjector-89769b4cb    1         1         1       4m58s
replicaset.apps/cert-manager-webhook-76bd54d4f       1         1         1       4m58s
replicaset.apps/cert-manager-69c6cb69f9              1         1         1       4m58s
root@BLMYCLDDL31451:/home/hcluser#
```

```
kubectl get apiservice
```

Returns two additional APIs:

```
v1.cert-manager.io
v1.acme.cert-manager.io
```

```
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get apiservice
NAME                                   SERVICE   AVAILABLE   AGE
v1.                                    Local     True        8m2s
v1.admissionregistration.k8s.io        Local     True        8m2s
v1.apiextensions.k8s.io                Local     True        8m2s
v1.authentication.k8s.io               Local     True        8m2s
v1.apps                                Local     True        8m2s
v1.authorization.k8s.io                Local     True        8m2s
v1.autoscaling                         Local     True        8m2s
v2.autoscaling                         Local     True        8m2s
v1.batch                               Local     True        8m2s
v1.certificates.k8s.io                 Local     True        8m2s
v1.coordination.k8s.io                 Local     True        8m2s
v1.discovery.k8s.io                    Local     True        8m2s
v1.events.k8s.io                       Local     True        8m2s
v1beta2.flowcontrol.apiserver.k8s.io   Local     True        8m2s
v1beta3.flowcontrol.apiserver.k8s.io   Local     True        8m2s
v1.networking.k8s.io                   Local     True        8m2s
v1.node.k8s.io                         Local     True        8m2s
v1.policy                              Local     True        8m2s
v1.rbac.authorization.k8s.io           Local     True        8m2s
v1.scheduling.k8s.io                   Local     True        8m2s
v1.storage.k8s.io                      Local     True        8m2s
v1beta1.storage.k8s.io                 Local     True        8m2s
v1.crd.projectcalico.org               Local     True        7m57s
v1.cert-manager.io                     Local     True        7m8s
v1.acme.cert-manager.io                Local     True        7m8s
```

Cert-manager: Custom Resource Definitions

Issuer: Namespaced resource that can issue certificates for the namespace where it resides.

ClusterIssuer: Non-namespaced resource that can issue global certificates for the entire cluster.

Certificate: Namespaced resource that defines X.509 certificate which will be automatically renewed. A Certificate references an Issuer or ClusterIssuer that determine what will be honoring the certificate request.

CertificateRequest: Namespaced resource in cert-manager that is used to request X.509 certificates from an Issuer. The CertificateRequest 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. CertificateRequests are typically consumed and managed by controllers or other systems and should not be used by humans - unless specifically needed.

https://cert-manager.io/docs/concepts/

SelfSigned Issuer

A Self-Signed Issuer creates Self-Signed Certificates in a specific namespace.

```yaml
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
  namespace: sandbox
spec:
  selfSigned: {}
```

```
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl create ns sandbox
namespace/sandbox created
root@BLMYCLDDL31451:/home/hcluser# nano self-signed-issuer.yaml
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl apply -f self-signed-issuer.yaml
issuer.cert-manager.io/selfsigned-issuer created
```

```
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get issuer -n sandbox
NAME                READY   AGE
selfsigned-issuer   True    20s
```

https://cert-manager.io/docs/configuration/selfsigned/

SelfSigned ClusterIssuer

A Self-Signed ClusterIssuer creates Self-Signed Certificates cluster-wide

```yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: selfsigned-cluster-issuer
spec:
  selfSigned: {}
```

```
root@BLMYCLDDL31451:/home/hcluser# nano self-signed-cluster-issuer.yaml
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl apply -f self-signed-cluster-issuer.yaml
clusterissuer.cert-manager.io/selfsigned-cluster-issuer created
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get clusterissuer
NAME                          READY   AGE
selfsigned-cluster-issuer     True    15s
```

https://cert-manager.io/docs/configuration/selfsigned/

Create a SelfSigned certificate to serve as root CA

SelfSigned issuers can be used to bootstrap a custom root certificate for a private CA.

This YAML will issue a root certificate and use that root as a CA issuer:

```
root@BLMYCLDDL31451:/home/hcluser# nano root-ca.yaml
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl apply -f root-ca.yaml
certificate.cert-manager.io/my-selfsigned-ca created
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get certificate -n sandbox
NAME               READY   SECRET        AGE
my-selfsigned-ca   True    root-secret   7s
```

```yaml
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-selfsigned-ca
  namespace: sandbox
spec:
  isCA: true
  commonName: my-selfsigned-ca
  secretName: root-secret
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-cluster-issuer
    kind: ClusterIssuer
    group: cert-manager.io
```

Describe the generated certificate

```
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl describe certificate -n sandbox
Name:         my-selfsigned-ca
Namespace:    sandbox
Labels:       <none>
Annotations:  <none>
API Version:  cert-manager.io/v1
Kind:         Certificate
Metadata:
```

If you describe the certificate, you can see all the events triggered and whether the issuance was successful.

```
Spec:
  Common Name:  my-selfsigned-ca
  Is CA:        true
  Issuer Ref:
    Group:  cert-manager.io
    Kind:   ClusterIssuer
    Name:   selfsigned-cluster-issuer
  Private Key:
    Algorithm:  ECDSA
    Size:       256
  Secret Name:  root-secret
Status:
  Conditions:
    Last Transition Time:  2023-01-18T18:03:59Z
    Message:               Certificate is up to date and has not expired
    Observed Generation:   1
    Reason:                Ready
    Status:                True
    Type:                  Ready
  Not After:               2023-04-18T18:03:59Z
  Not Before:              2023-01-18T18:03:59Z
  Renewal Time:            2023-03-19T18:03:59Z
  Revision:                1
Events:
  Type    Reason     Age   From                                       Message
  ----    ------     ----  ----                                       -------
  Normal  Issuing    43s   cert-manager-certificates-trigger          Issuing certificate as Secret does not exist
  Normal  Generated  43s   cert-manager-certificates-key-manager      Stored new private key in temporary Secret resource "my-selfsigned-ca-jsmc2"
  Normal  Requested  43s   cert-manager-certificates-request-manager  Created new CertificateRequest resource "my-selfsigned-ca-4jggc"
  Normal  Issuing    43s   cert-manager-certificates-issuing          The certificate has been successfully issued
```

```
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get secret -A
NAMESPACE      NAME                      TYPE                 DATA   AGE
cert-manager   cert-manager-webhook-ca   Opaque               3      50m
sandbox        root-secret               kubernetes.io/tls    3      18m
```

CA Issuer

The CA issuer represents a Certificate Authority whose certificate and private key are stored inside the cluster as a Kubernetes Secret.

Certificates issued by a CA issuer will not be publicly trusted.

CA Issuers must be configured with a certificate and private key stored in a Kubernetes secret. You can create a root certificate using a SelfSigned issuer as in previous slide.

Your certificate's secret should reside in the same namespace as the Issuer, or otherwise in the Cluster Resource Namespace in the case of a ClusterIssuer.

The Cluster Resource Namespace is defaulted as being the cert-manager namespace, but can be configured using the --cluster-resource-namespace flag on the cert-manager controller.

```
apiVersion: cert-
manager.io/v1
kind: Issuer
metadata:
  name: my-ca-issuer
  namespace: sandbox
spec:
  ca:
    secretName: root-secret
```

```
root@BLMYCLDDL31451:/home/hcluser# nano ca-issuer.yaml
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl apply -f ca-issuer.yaml
issuer.cert-manager.io/my-ca-issuer created
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get issuer -n sandbox
NAME                READY   AGE
selfsigned-issuer   True    34m
my-ca-issuer        True    31s
```

https://cert-manager.io/docs/configuration/ca/

CA ClusterIssuer

The CA issuer represents a Certificate Authority whose certificate and private key are stored inside the cluster as a Kubernetes Secret.

Certificates issued by a CA issuer will not be publicly trusted.

CA Issuers must be configured with a certificate and private key stored in a Kubernetes secret. You can create a root certificate using a SelfSigned issuer as in previous slide.

Your certificate's secret should reside in the same namespace as the Issuer, or otherwise in the Cluster Resource Namespace in the case of a ClusterIssuer.

The Cluster Resource Namespace is defaulted as being the cert-manager namespace, but can be configured using the --cluster-resource-namespace flag on the cert-manager controller.

```yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: my-ca-cluster-issuer
spec:
  ca:
    secretName: root-secret
```

```
root@BLMYCLDDL31451:/home/hcluser# nano ca-issuer.yaml
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl apply -f ca-issuer.yaml
issuer.cert-manager.io/my-ca-issuer created
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get issuer -n sandbox
NAME                READY   AGE
selfsigned-issuer   True    34m
my-ca-issuer        True    31s
```

https://cert-manager.io/docs/configuration/ca/

Configure ingress with a certificate signed by CA Issuer

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    # add an annotation indicating the issuer to use.
    cert-manager.io/issuer: my-ca-issuer
    cert-manager.io/common-name: mydomain.com
  name: my-ingress
  namespace: sandbox
spec:
  rules:
  - host: mydomain.com
    http:
      paths:
      - pathType: Prefix
        path: /
        backend:
          service:
            name: myservice
            port:
              number: 80
  tls: # < placing a host in the TLS config will determine what ends up in the cert's subjectAltNames
  - hosts:
    - mydomain.com
    secretName: myingress-cert # < cert-manager will store the created certificate in this secret.
```

For a local test, add: 127.0.0.1 mydomain.com to /etc/hosts

https://cert-manager.io/docs/usage/ingress/

Verify that the certificate was created for Ingress

```
root@BLMYCLDDL31451:/home/hcluser# nano my-ingress.yaml
root@BLMYCLDDL31451:/home/hcluser# microk8s enable ingress
Infer repository core for addon ingress
Enabling Ingress
ingressclass.networking.k8s.io/public created
ingressclass.networking.k8s.io/nginx created
namespace/ingress created
serviceaccount/nginx-ingress-microk8s-serviceaccount created
clusterrole.rbac.authorization.k8s.io/nginx-ingress-microk8s-clusterrole created
role.rbac.authorization.k8s.io/nginx-ingress-microk8s-role created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress-microk8s created
rolebinding.rbac.authorization.k8s.io/nginx-ingress-microk8s created
configmap/nginx-load-balancer-microk8s-conf created
configmap/nginx-ingress-tcp-microk8s-conf created
configmap/nginx-ingress-udp-microk8s-conf created
daemonset.apps/nginx-ingress-microk8s-controller created
Ingress is enabled
```

```
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl apply -f my-ingress.yaml
ingress.networking.k8s.io/my-ingress created
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get secrets -n sandbox
NAME             TYPE                DATA   AGE
root-secret      kubernetes.io/tls   3      38m
myingress-cert   kubernetes.io/tls   3      18s
```

```
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get secret myingress-cert -n sandbox -o yaml
apiVersion: v1
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUJkakNDQVJ5Z0F3SUJBZ0lRYnE3MU52TnVrdHlwbUtvU2t
HekVaTUJjR0ExVUVBeE1RYlhrdGMyVnNabk5wWjI1bFpDMWpZVEJaTUJNR0J5cUdTTTQ5CkFnRUdDQ3FHU000OUF3RUhBMElB
L3dRRQpBd0lDcERBUEJnTlZIUk1CQWY4RUJUQURBUUgvTUIwR0ExVWREZ1FXQkJSVDNRZTRSUmpQWjkwd1A3WER3TU03Ci9yR
0VXUllvRXZJd0VER0QKLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo=
  tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNWekNDQWZ5Z0F3SUJBZ0lSQU1PVzR1YlExb2hjekRuSF
FNQmN4RlRBVEJnTlZCQU1UREcxNVpHOXRZV2x1TG1OdmJUQ0NBU0l3RFFZSktvWklodmNOCkFRRUJCUUFEZ2dFUEFEQ0NBUW9
MQU10RApwOElMb3dNVktvdDZaVkVSRHRsR2dDR3lPejNFRWdLK3B3OC9hT2FSTXBBVjFtRjl0VFRQT1YyRExueU9TYUlFCnZT
c1htZ1pHRGFGM3hPSGl4d1pIM0RGTgo1MWNVV0J6TE1adUFpM3p6eVhvcGZMMjhCOGlrU2hSYi82bnFkQXZBTm56VkxIVUNBd
kJBd0RvSU1iWGxrYjIxaGFXNHVZMjl0TUFvR0NDcUdTTTQ5QkFNQwpBMGtBTUVZQ0lRRGRNV2RDNzVyU1h3N05zSFA3TXMvUw
  tls.key: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcEFJQkFBS0NBUUVBdnpHeGNQdXRJdVc1ZVNabU
8yVWFBSWJJN1BjUVNBcjZuRHo5bzVwRXlrQlhXWgyMU5NODVYWU11Zkk1Sm9nUzlJOE1iWVVNZAo4Y3oxY2hSMEtOamlpaDk
uVnhSWUhNc3gKbTRDTGZQUEplaWw4dmJ3SHlLUktGRnYvcWVwMEM4QTJmTlVzZFFJREFRQUJBb0lCQUU1YzRycEpIUXhabGF4
aWxFaTN4bkdDek9WNjJkeDlTM3d2NzFGWDAKR3dJOVFSSFNRU3YwNXpFeHJDTDIrcTBDUGpkaWNiYzdyWTFoeFdEWG9kMFltV
lowVThXQkRhVzFkUE1RenBBVXlicUJwdmkraWtEVGdYamJqNWpRekF5THIKVWMybGNiMENnWUVBMFAxR0Q0ZDI0akRjYnR5Vm
c2V3dVc2tjcURvRTJWWGRIRFZpQVAxZFl5RlRTbUVlTXlVUjU4a3RsbXlINklMUmltZzhDZ1lFQTZqT3MKcGkzaGdtZFhSaTg
vTnp5aXlyVjJlYXdqCnBUUUlLUmZsTVdycENqOXlaQm4ra2dXUlI2cW9DbGxKVGVZd3BUc0NnWUE5V3AyKzRWNGRlWXE5WVRU
dXNZM2FRQUNSd3JhNXQyTjZiSkdoQ01EcDRWYTZkCjFpT1cwdXVhSGxodCtzT2I1bGtzTVFLQmdRQy8vb2pFMnN3S09XbWtrY
1M4Y3UwckJYeTIyNVZWN2MwSlZYYzk0U2FxdHlnR2U2bWgxa0xiZXg1cG1rWGUwCnNzYWlUUUtCZ1FETm9OSkRvZnZicm9FRn
phL1FLM0p5TndIcTF2NmFUSHcwOXZ5N2RkQmxIQmZyUXJXaFUwK002TmhQNVRjOFo0QnVEWGc9PQotLS0tLUVORCBSU0EgUFJ
kind: Secret
metadata:
  annotations:
    cert-manager.io/alt-names: mydomain.com
    cert-manager.io/certificate-name: myingress-cert
    cert-manager.io/common-name: mydomain.com
    cert-manager.io/ip-sans: ""
    cert-manager.io/issuer-group: cert-manager.io
    cert-manager.io/issuer-kind: Issuer
    cert-manager.io/issuer-name: my-ca-issuer
    cert-manager.io/uri-sans: ""
  creationTimestamp: "2023-01-18T18:56:33Z"
  name: myingress-cert
  namespace: sandbox
  resourceVersion: "8325"
  uid: a470b101-65c5-4d3e-8fba-2ca270f55e74
type: kubernetes.io/tls
```

```
root@BLMYCLDDL31451:/home/hcluser# microk8s kubectl get Certificate -n sandbox -o wide
NAME              READY   SECRET           ISSUER                     STATUS                                          AGE
my-selfsigned-ca  True    root-secret      selfsigned-cluster-issuer  Certificate is up to date and has not expired   42m
myingress-cert    True    myingress-cert   my-ca-issuer               Certificate is up to date and has not expired   3m41s
```

Verify the TLS connection to Ingress using openssl

You can use the following command:

```
openssl s_client –showcerts –connect mydomain.com:443
```

to verify the actual parameters of the TLS connection between a client and your Ingress Resource.

This confirms the certificate being used and the issuer, but it fails verification because the self signed issuer (my-selfsigned-ca) is not trusted.

```
root@BLMYCLDDL31451:/home/hcluser# openssl s_client -showcerts -connect mydomain.com:443
CONNECTED(00000003)
depth=0 CN = mydomain.com
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 CN = mydomain.com
verify error:num=21:unable to verify the first certificate
verify return:1
---
Certificate chain
 0 s:CN = mydomain.com
   i:CN = my-selfsigned-ca
-----BEGIN CERTIFICATE-----
MIICVzCCAfygAwIBAgIRAMOW4ubQ1ohczDnHYyXx6KwwCgYIKoZIzj0EAwIwGzEZ
MBcGA1UEAxMQbXktc2VsZnNpZ25lZC1jYTAeFw0yMzAxMTgxODU2MzNaFw0yMzA0
MTgxODU2MzNaMBcxFTATBgNVBAMTDG15ZG9tYWluLmNvbTCCASIwDQYJKoZIhvcN
AQEBBQADggEPADCCAQoCggEBAL8xsXD7rSLluXkmZggJEp1Y6v8OG9OMQMroIaVz
QxFbohenqUEFgK2u4VHt6WSw+zZiGp/rc6/SRyd6PQLC4TfxjEUgyr3BaroLAMtD
p8ILowMVKot6ZVERDtlGgCGyOz3EEgK+pw8/aOaRMpAV1mF9tTTPOV2DLnyOSaIE
vSPDG2FDHfHM9XIUdCjY4oofdBC56JhFt8oWJK4v9ZwI2mo3cKJciVQ06xAKLFtN
9D/4loKglfBK5HtfzJzhjkaPuJoWtc0+ruESTT4lmqsXmgZGDaF3xOHixwZH3DFN
51cUWBzLMZuAi3zzyXopfL28B8ikShRb/6nqdAvANnzVLHUCAwEAAaNaMFgwDgYD
VR0PAQH/BAQDAgWgMAwGA1UdEwEB/wQCMAAwHwYDVR0jBBgwFoAUU90HuEUYz2fd
MD+1w8DDO/6xOyAwFwYDVR0RBBAwDoIMbXlkb21haW4uY29tMAoGCCqGSM49BAMC
A0kAMEYCIQDdMWdC75rSXw7NsHP7Ms/Qg4RDSkyzLKCfNHvjxVT/YwIhAKswFOw0
E1Qht+JWWBYc41OcYSvTarwrub6gz+lQ3CdI
-----END CERTIFICATE-----
---
Server certificate
subject=CN = mydomain.com

issuer=CN = my-selfsigned-ca

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 1163 bytes and written 384 bytes
Verification error: unable to verify the first certificate
```

HCLSoftware

hcltechsw.com