

# **Exercise:Camunda Self-Managed Installation**

## **Step 1 — Installing Docker**

First, update your existing list of packages:

```
$ sudo apt update
```

Next, install a few prerequisite packages which let `apt` use packages over HTTPS:

```
$ sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

Then add the GPG key for the official Docker repository to your system:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Add the Docker repository to APT sources:

```
$ sudo add-apt-repository "deb [arch=amd64]  
https://download.docker.com/linux/ubuntu focal stable"
```

This will also update our package database with the Docker packages from the newly added repo.

Make sure you are about to install from the Docker repo instead of the default Ubuntu repo:

```
$ apt-cache policy docker-ce
```

You'll see output like this, although the version number for Docker may be different:

```
$ docker-ce:
```

```
Installed: (none)
```

```
Candidate: 5:19.03.9~3-0~ubuntu-focal
```

```
Version table:
```

```
5:19.03.9~3-0~ubuntu-focal 500
```

```
500 https://download.docker.com/linux/ubuntu focal/stable amd64 Packages
```

Finally, install Docker:

```
$ sudo apt install docker-ce
```

Docker should now be installed, the daemon started, and the process enabled to start on boot. Check that it's running:

```
$ sudo systemctl status docker
```

The output should be similar to the following, showing that the service is active and running:

Output● docker.service - Docker Application Container Engine

Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)

Active: active (running) since Tue 2020-05-19 17:00:41 UTC; 17s ago

TriggeredBy: ● docker.socket

Docs: <https://docs.docker.com>

Main PID: 24321 (dockerd)

Tasks: 8

Memory: 46.4M

CGroup: /system.slice/docker.service

└─24321 /usr/bin/dockerd -H fd:// --  
containerd=/run/containerd/containerd.sock

add your username to the `docker` group:

```
$ sudo usermod -aG docker ${USER}
```

To apply the new group membership, log out of the server and back in, or type the following:

```
$ su - ${USER}
```

You will be prompted to enter your user's password to continue.

Confirm that your user is now added to the **docker** group by typing:

```
$ groups
```

Once Docker is installed, you need to install `kind` along with three other packages.

## Step 2 — Installing Kind

```
$ [ $(uname -m) = x86_64 ] && curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.25.0/kind-linux-amd64
```

```
$ chmod +x ./kind
```

```
$ sudo mv ./kind /usr/local/bin/kind
```

```
$ kind --help
```

### Step 3 — Installing Kubectl

Install kubectl binary with curl on Linux

#### **Download the latest release with the command:**

```
$ curl -LO https://dl.k8s.io/release/\$\(curl -L -s https://dl.k8s.io/release/stable.txt\)/bin/linux/amd64/kubectl
```

Validate the binary (optional)

Download the kubectl checksum file:

```
$ curl -LO https://dl.k8s.io/release/\$\(curl -L -s https://dl.k8s.io/release/stable.txt\)/bin/linux/amd64/kubectl.sha256
```

Validate the kubectl binary against the checksum file:

```
$ echo "$(cat kubectl.sha256) kubectl" | sha256sum --check
```

If valid, the output is:

```
$ kubectl: OK
```

If the check fails, sha256 exits with nonzero status and prints output similar to:

```
kubectl: FAILED
```

```
sha256sum: WARNING: 1 computed checksum did NOT match
```

Install kubectl

```
$ sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Test to ensure the version you installed is up-to-date:

```
$ kubectl version --client
```

## Step 4 — Installing Helm

### From Script

Helm now has an installer script that will automatically grab the latest version of Helm and [install it locally](#).

You can fetch that script, and then execute it locally. It's well documented so that you can read through it and understand what it is doing before you run it.

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
```

```
$ chmod 700 get_helm.sh
```

```
$ ./get_helm.sh
```

```
$ helm version
```

## Step 5 — Installing k9s (optional)

### Step 1: Download the K9s Debian Package

First, you need to download the latest K9s Debian package. Open your terminal and run the following command:

```
$ wget https://github.com/derailed/k9s/releases/download/v0.32.5/k9s_linux_amd64.deb
```

This command fetches the Debian package for K9s version 0.32.5. You can check for the latest version on the K9s GitHub releases page [<https://github.com/derailed/k9s/releases>] and replace the version in the URL if necessary.

## Step 2: Install K9s

Once the download is complete, you can install K9s using the following command:

```
$ sudo apt install ./k9s_linux_amd64.deb
```

This command installs the K9s package on your system. The ./ before the package name indicates that the file is in the current directory.

## Step 3: Clean Up

After installation, you can remove the downloaded .deb file to save space:

```
$ rm k9s_linux_amd64.deb
```

This step is optional but recommended if you want to keep your system clean.

## Step 4: Run K9s

Now that K9s is installed, you can start it by simply typing:

```
$ k9s
```

## Configuring Camunda Self-Managed

While there are other ways to deploy Camunda Self-Managed, we recommend using our Helm charts. Helm charts provide a single place to configure every aspect of your deployment: what services you want to install and how those services are configured.

There are many different ways to configure these services:

- With or without SSL/TLS;
- With an ingress controller or using port forwarding;
- Connecting to an existing Elasticsearch or Keycloak instance or using a new one;
- Using an existing Postgres database or using a new one;

- With or without multi-tenancy;
- ... And much more!

Helm charts accept a values file, in YAML format just like the charts themselves, which allows users to configure the services. Not every value can be overridden: it depends on how the original charts were built.

Instead, let's review a bare minimum configuration.

```
## Camunda Helm chart.
```

```
#
```

```
global:
```

```
  # https://github.com/camunda/camunda-platform/releases
```

```
  # https://hub.docker.com/u/camunda
```

```
  image:
```

```
    tag:
```

```
identity:
```

```
  auth:
```

```
    # Disable Identity authentication for local development
```

```
    # it will fall back to basic-auth: demo/demo as default user
```

```
    enabled: false
```

```
optimize:
```

```
  enabled: true
```

```
console:
```

```
  # Camunda Enterprise repository.
```

```
  # https://hub.docker.com/r/camunda/console/tags
```

```
  image:
```

```
    tag: 8.6.10
```

```
connectors:
```

```
  # https://hub.docker.com/r/camunda/connectors-bundle/tags
```

```
  enabled: true
```

```
  inbound:
```

mode: "disabled"

image:

repository: camunda/connectors-bundle

tag: 8.6.4

operate:

# <https://hub.docker.com/r/camunda/operate/tags>

image:

tag: 8.6.3

optimize:

# <https://hub.docker.com/r/camunda/optimize/tags>

image:

repository: camunda/optimize

tag: 8.6.2

tasklist:

# <https://hub.docker.com/r/camunda/tasklist/tags>

image:

repository: camunda/tasklist

tag: 8.6.5

webModeler:

# <https://hub.docker.com/r/camunda/web-modeler-restapi>

image:

# renovate: datasource=docker depName=camunda/web-modeler-restapi

tag: 8.6.3

zeebe:

# <https://hub.docker.com/r/camunda/zeebe/tags>

clusterSize: 1

partitionCount: 1

replicationFactor: 1

pvcSize: 10Gi

resources: {}

initResources: {}

image:

repository: camunda/zeebe

tag: 8.6.5

zeebeGateway:

# <https://hub.docker.com/r/camunda/zeebe/tags>

replicas: 1

image:

repository: camunda/zeebe

tag: 8.6.5

#

# Identity

#

identity:

# <https://hub.docker.com/r/camunda/identity/tags>

enabled: false

image:

repository: camunda/identity

tag: 8.6.5

identityKeycloak:

# <https://hub.docker.com/r/bitnami/keycloak/tags>

enabled: false

image:

repository: bitnami/keycloak

tag: 25.0.6

postgresql:

# <https://hub.docker.com/r/bitnami/postgresql/tags>

image:

repository: bitnami/postgresql

tag: 15.9.0-debian-12-r0



```

#
# Elasticsearch
#

elasticsearch:
  # https://hub.docker.com/r/bitnami/elasticsearch/tags
  resources: {}
  initResources: {}
  replicas: 1
  minimumMasterNodes: 1
  # Allow no backup for single node setups
  clusterHealthCheckParams: "wait_for_status=yellow&timeout=1s"
  image:
    repository: bitnami/elasticsearch
    tag: 8.15.4

volumeClaimTemplate:
  accessModes: [ "ReadWriteOnce" ]
  storageClassName: "standard"
  resources:
    requests:
      storage: 15Gi

```

## Starting Camunda Self-Managed

Copy the configuration file above and save it as `camunda-values.yaml`. Next, open a terminal or command prompt and navigate to where you just saved the `camunda-values.yaml` file.

The first step to starting all the services is to create a *cluster*.

A Kubernetes cluster is a set of nodes, and each node can run one or more pods, and a pod can run one or more containers. This architecture is what makes Kubernetes highly scalable (and also what can make it very confusing and complex). For our simple local installation, we need a cluster with minimum 3 nodes. You can create the cluster by using the following configuration.

```
# this config file contains all config fields with comments
```

# NOTE: this is not a particularly useful config file

kind: Cluster

apiVersion: kind.x-k8s.io/v1alpha4

# patch the generated kubeadm config with some extra settings

kubeadmConfigPatches:

- |

apiVersion: kubelet.config.k8s.io/v1beta1

kind: KubeletConfiguration

evictionHard:

nodefs.available: "0%"

# patch it further using a JSON 6902 patch

kubeadmConfigPatchesJSON6902:

- group: kubeadm.k8s.io

version: v1beta3

kind: ClusterConfiguration

patch: |

- op: add

path: /apiServer/certSANs/-

value: my-hostname

# 1 control plane node and 3 workers

nodes:

# the control plane node config

- role: control-plane

#- role: control-plane

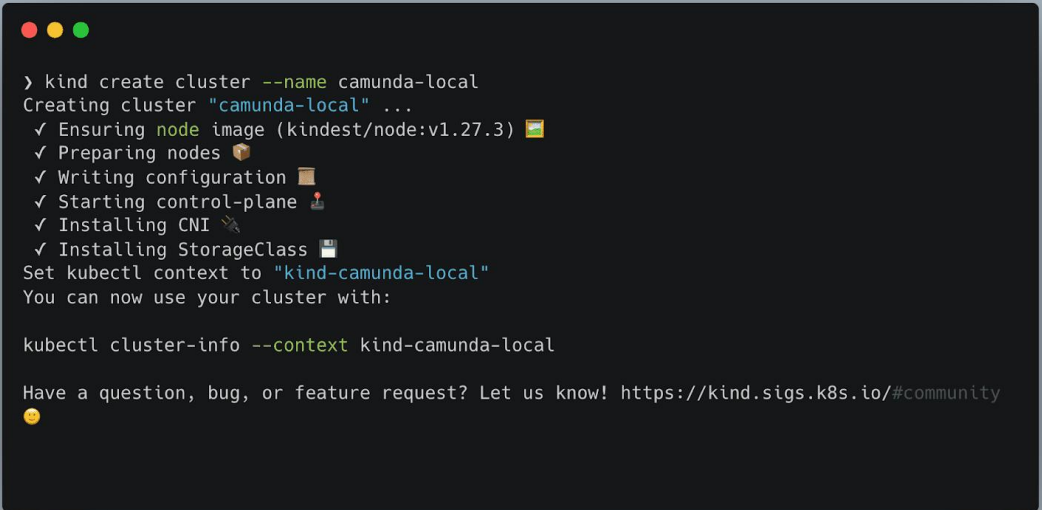
# the two workers

- role: worker
- role: worker
- role: worker

Copy the above configuration into cluster-config.yaml file.

Now create a cluster using `kind`. In your terminal, run the following command:

```
$ kind create cluster --name camunda-local --config cluster-config.yaml
```



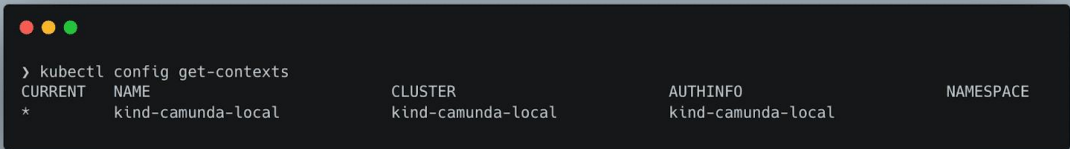
```
> kind create cluster --name camunda-local
Creating cluster "camunda-local" ...
 ✓ Ensuring node image (kindest/node:v1.27.3)
 ✓ Preparing nodes
 ✓ Writing configuration
 ✓ Starting control-plane
 ✓ Installing CNI
 ✓ Installing StorageClass
Set kubectl context to "kind-camunda-local"
You can now use your cluster with:

kubectl cluster-info --context kind-camunda-local

Have a question, bug, or feature request? Let us know! https://kind.sigs.k8s.io/#community
```

This tells `kind` to create a new cluster with a specific context name and configuration. Kubernetes contexts allow you to easily switch between different Kubernetes instances, which is helpful if you are running more than one application locally with Kubernetes. To see all the available contexts, you run the following command:

```
$ kubectl config get-contexts
```



```
> kubectl config get-contexts
CURRENT  NAME           CLUSTER           AUTHINFO           NAMESPACE
*        kind-camunda-local  kind-camunda-local  kind-camunda-local
```

(You may have noticed that you have a context named `kind-camunda-local` instead of just `camunda-local`. `kind` prefixes the contexts it creates.)

The next step is to tell Kubernetes that we want to use our newly created context and cluster:

```
$ kubectl config use-context kind-camunda-local
```

Our cluster is now configured, and the tools are set to use the `kind-camunda-local` context by default. It's time to use Helm to start all of our services. First, we add the [Camunda Helm repository](#) so that Helm knows where to fetch the configuration files from:

```
$ helm repo add camunda https://helm.camunda.io
```

```
$ helm repo update
```

Then, we tell Helm to install everything by applying our `camunda-values.yaml` file to the official Camunda Helm charts. Because we set the default Kubernetes context to our new cluster already, everything will be installed to that cluster without needing to explicitly specify it.

```
$ helm install camunda-platform camunda/camunda-platform -f camunda-values.yaml
```

This command tells Helm to use our `camunda-values.yaml` file to configure the Helm charts pulled from the `camunda/camunda-platform` repository, and gives it the release name `camunda-platform`. ([Read more about Helm release names here.](#))

That's it! Everything is starting up behind the scenes. This is where `k9s` comes in handy. In your terminal, run the command `k9s` to open the application. After it initializes you should see a screen that looks quite similar to this:



Before we get into why, let's hide the `kube-system` pods: these pods are essentially system pods, related to running Kubernetes and not directly related to running Camunda. In `k9s` press the number 1 key on your keyboard and it will filter the list down to just the Camunda services.

```

Context: kind-camunda-local
Cluster: kind-camunda-local
User: kind-camunda-local
K8s Rev: v0.27.4
K8s Rev: v1.27.3
CPU: n/a
MEM: n/a

<0> all
<1> default
<0> Attach
<ctrl-d> Delete
<0> Describe
<0> Edit
<?> Help
<ctrl-k> Kill
<0> Logs
<0> Previous
<shift-f> Port-Forward
<0> Shell
<0> Show Node
<0> Show PortForward

NAME
camunda-platform-connectors-5fccd6db76-62qp2
camunda-platform-operate-5f6fcd7b7c-w02z9
camunda-platform-tasklist-d5b85b946-mpvgd
camunda-platform-zeebe-0
camunda-platform-zeebe-gateway-b57586b5-mqttc
elasticsearch-master-0

PF
●
●
●
●
●
●

READY
0/1
0/1
0/1
0/1
0/1
0/1

RESTARTS
0
0
0
0
0
0

STATUS
Running
Running
ContainerCreating
ContainerCreating
ContainerCreating
Running

IP
10.244.0.20
10.244.0.19
n/a
n/a
n/a
10.244.0.22

NODE
camunda-local-control-plane
camunda-local-control-plane
camunda-local-control-plane
camunda-local-control-plane
camunda-local-control-plane
camunda-local-control-plane

AGE
32s
32s
32s
32s
32s
32s

<pod>

```

That's better! At this point the pods should all be starting up. This can take some time. Depending on your system resources, internet connection speed, and other factors, it may take several minutes for everything to start.

(Don't start worrying until at least five minutes have gone by!)

The names start with `camunda-platform` (which is one of the parameters we used in the `helm install` command), followed by a service name, followed by a unique identifier. (You might notice that Zeebe and Elasticsearch just have `0` as a suffix rather than a unique identifier. That is because these two services are [StatefulSets](#), the Kubernetes term for services that hold their state. You can [learn more about StatefulSets here](#)!)

After a few minutes, all of your services should have a Status of `Running`, and you should see `1/1` under the `Ready` column.

```
Context: kind-camunda-local
Cluster: kind-camunda-local
User: kind-camunda-local
K9s Rev: v0.27.4
K8s Rev: v1.27.3
CPU: n/a
MEM: n/a

<0> all <a> Attach <l> Logs
<1> default <ctrl-d> Delete <p> Logs Previous
<d> Describe <shift-f> Port-Forward
<e> Edit <s> Shell
<?> Help <m> Show Node
<ctrl-k> Kill <f> Show PortForward

Pod(s) (default) [6]
NAME PF READY RESTARTS STATUS IP NODE AGE
camunda-platform-connectors-5fccd6db76-62qp2 ● 1/1 0 Running 10.244.0.20 camunda-local-control-plane 64m
camunda-platform-operate-5f6fcd7b7c-wb2s9 ● 1/1 1 Running 10.244.0.19 camunda-local-control-plane 64m
camunda-platform-tasklist-d5b85b946-mpvgd ● 1/1 0 Running 10.244.0.24 camunda-local-control-plane 64m
camunda-platform-zeebe-0 ● 1/1 0 Running 10.244.0.25 camunda-local-control-plane 64m
camunda-platform-zeebe-gateway-b57586b5-mqttc ● 1/1 0 Running 10.244.0.21 camunda-local-control-plane 64m
elasticsearch-master-0 ● 1/1 0 Running 10.244.0.22 camunda-local-control-plane 64m
```

## What if a service doesn't start?

Well ... it depends! There are lots of reasons that a service may not start. Because we are using a very minimal configuration, the first thing to check would be resources. Inside Docker, you can allocate more CPU or memory for the containers (refer to the Docker documentation for your version of Docker on how to do this).

## What's Next?

First, *congratulations!* 🎉 You now have a running, fully functional Camunda Self-Managed installation on your workstation!

## Port forwarding

You probably noticed that you couldn't connect to any of the services.

It's important to think about your cluster as a separate network, even though it's installed on your local workstation rather than in the cloud. Whether you start a single Docker container, or you build a local Kubernetes cluster, the effect is the same: that containerized service will be running on a virtual network. You need to tell both the cluster and your workstation how they can talk to one another.

There are two ways of doing this with Kubernetes: port forwarding, and using an ingress controller.

Port forwarding, sometimes referred to as "port mapping," is the most basic solution. Keen eyed users may have noticed the output of the `helm install` command contains this:

```
Operate:
> kubectl port-forward svc/camunda-platform-operate 8081:80
Tasklist:
> kubectl port-forward svc/camunda-platform-tasklist 8082:80

Connectors:
> kubectl port-forward svc/camunda-platform-connectors 8088:8080
```

If you want to access one of those services, simply copy and paste the command! Let's use this command for Operate as an example: `kubectl port-forward svc/camunda-platform-operate 8081:80`. The Operate service is listening on port 80 (the port is configurable in the [Helm values.yaml file](#) if you wish to change it). Behind the scenes, `kubectl` is telling Kubernetes to listen on the first port ("8081") and send the network traffic to the second port ("80") *inside the cluster*.

It's as simple as that! There is one important thing to remember when using the `kubectl port-forward` command: the command doesn't return, which means your terminal will not return to a prompt. If you want to forward multiple ports, you will need to open multiple terminal windows or write a custom script.

## Ingress controllers

"An Ingress controller abstracts away the complexity of Kubernetes application traffic routing and provides a bridge between Kubernetes services and external ones."

In other words, instead of manually configuring all the routes needed for your inbound traffic to get to the right services inside your cluster, the ingress controller handles it automatically. Ingress controllers also act as load balancers, routing traffic evenly across your distributed services.

There are several different ingress controllers you can choose for your local deployment. Which one you choose depends on a number of factors, including the environment you are deploying it to. We will be using the `ingress-nginx` package for this exercise.

If you are getting ready to deploy to the cloud or a different Kubernetes environment, be sure to check their documentation. Many cloud providers offer their own ingress controllers that are better suited and easier to configure for those environments.

`kind` requires a small amount of additional configuration to make the ingress work. When creating your cluster you need to provide a configuration file. If you have already created a cluster, you will need to delete it using the `kind delete cluster --name camunda-local` command.

First, create a new file name `kind.config` with the following contents:

```
kind: Cluster
```

```
apiVersion: kind.x-k8s.io/v1alpha4
```



nodes:

- role: control-plane

kubeadmConfigPatches:

- |

kind: InitConfiguration

nodeRegistration:

kubeletExtraArgs:

node-labels: "ingress-ready=true"

extraPortMappings:

- containerPort: 80

hostPort: 80

- containerPort: 443

hostPort: 443

- containerPort: 26500

hostPort: 26500

Then, recreate the cluster using `kind create cluster --name camunda-local --config kind.config`, then deploy the Helm charts again with the same `helm install camunda-platform camunda/camunda-platform -f camunda-values.yaml`.

Finally, run the following command to install the ingress controller:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/kind/deploy.yaml.
```

(For more information about using `kind` with ingress controllers, refer to [their documentation!](#))

Now that we have an ingress controller we need to configure Camunda's services to work with the ingress. (More specifically, we need to configure the pods the services are running in to work with the ingress.)

## Combined or separated ingress?

There are two ways to configure the ingress: combined or separated.

A combined ingress configuration uses the same domain for all the services, and routes based on the path. For instance, Identity would be available at `https://domain.com/identity`, Operate would be available at `https://domain.com/operate`, and so on. When using a separated ingress, each service is available on its own domain. For instance, Identity would be available at `http://identity.domain.com/`, Operate would be available at `https://operate.domain.com/`, and so on.

For this exercise we will use the combined configuration. However, there is one quirk with this particular setup to be aware of! Zeebe Gateway uses gRPC, which uses HTTP/2. This means that Zeebe Gateway cannot be on a path. (Reason is because the URL `https://domain.com/zeebe-gateway/` uses HTTP and not HTTP/2.)

Note: If you're interested in using a separated setup, you can [review our guide in the docs!](#)

With that in mind, let's look at the changes new `camunda-values.yaml` file:

global:

ingress:

enabled: true

className: nginx

host: "camunda.local"

operate:

contextPath: "/operate"

tasklist:

contextPath: "/tasklist"

zeebe-gateway:

ingress:

enabled: true

className: nginx

host: "zeebe.camunda.local"

The changes are pretty straightforward. Globally, we enable the ingress and give it a `className` of "nginx" because we are using the `ingress-nginx` controller. (If you are using a different controller, the `className` may be different, check the controllers documentation!) We also define the host: this is the domain that all the paths will use. For this example, I am using "camunda.local", but you can use any domain name that doesn't conflict with any other domain name. For Operate and Tasklist, we define what the path is. Last, for Zeebe Gateway, we define separate ingress using the subdomain "zeebe.camunda.local".

The domain "camunda.local" doesn't exist, which means that your workstation doesn't know how to connect to it. You will need to add two entries to your workstation's hosts file that resolve "camunda.local" and "zeebe.camunda.local" (or whatever domain you chose) to the IP address "127.0.0.1".

## Configuring TLS/SSL

The last step to get everything working is to generate a certificate and secure the ingress with it. While Camunda does not require TLS to work, Nginx does require a certificate for HTTP/2. There are many ways to generate a certificate, but for simplicity we will use a self-signed certificate.

To generate a certificate, execute the following command. You will be asked a series of questions to configure the certificate: for this example, the values you enter do not matter.

```
$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -sha256 -days 365 --nodes -addext 'subjectAltName=DNS:camunda.local'
```

I won't cover all the parameters here, but there are four important values:

- The `-days` parameter sets how long the certificate is valid for; in this example, it will expire in 1 year.
- The `-keyout` parameter configures the file name of the private key file that the certificate is signed with. You will need this key to install the certificate.
- The `-out` parameter configures the file name of the certificate itself.
- The `-addext` parameter configures the domain that this certificate is valid for. Because I configured our ingress to use "camunda.local", that is the domain used for this certificate.

However, we had to configure a separate ingress for Zeebe Gateway, which needs its own certificate. The command is nearly the same: just change the file names and the domain!

```
$ openssl req -x509 -newkey rsa:4096 -keyout key-zeebe.pem -out cert-zeebe.pem -sha256 -days 365 --nodes -addext 'subjectAltName=DNS:zeebe.camunda.local'
```

Next, we need to add the certificates to our Kubernetes clusters as [Secrets](#). Secrets are how Kubernetes saves sensitive information that shouldn't be available in plaintext files like the values.yaml file. Instead, the values.yaml file references the secret name and Kubernetes handles the rest. We will need to create two secrets, one for each certificate:

```
$ kubectl create secret tls tls-secret --cert=cert.pem --key=key.pem
```

```
$ kubectl create secret tls tls-secret-zeebe --cert=cert-zeebe.pem --key=key-zeebe.pem
```

Finally, we need to configure TLS in our values.yaml file, using the secret names we just created. The complete file, with the combined ingress and TLS configured, looks like this:

```
global:
```

```
  ingress:
```

```
    enabled: true
```

```
    className: nginx
```

```
    host: "camunda.local"
```

```
    tls:
```

```
      enabled: true
```

```
      secretName: "tls-secret"
```

```
  identity:
```

```
    auth:
```

```
      # Disable Identity authentication for local development
```

```
      # it will fall back to basic-auth: demo/demo as default user
```

```
      enabled: false
```

```
# Disable Identity for local development
```

```
identity:
```

```
  enabled: false
```

# Disable Optimize

optimize:

enabled: false

operate:

contextPath: "/operate"

tasklist:

contextPath: "/tasklist"

# Reduce resource usage for Zeebe and Zeebe-Gateway

zeebe:

clusterSize: 1

partitionCount: 1

replicationFactor: 1

pvcSize: 10Gi

resources: {}

initResources: {}

zeebe-gateway:

replicas: 1

ingress:

enabled: true

className: nginx

host: "zeebe.camunda.local"

tls:

enabled: true

secretName: "tls-secret-zeebe"

# Enable Outbound Connectors only

connectors:

enabled: true

inbound:

mode: "disabled"

# Configure Elasticsearch to make it running for local development

elasticsearch:

resources: {}

initResources: {}

replicas: 1

minimumMasterNodes: 1

# Allow no backup for single node setups

clusterHealthCheckParams: "wait\_for\_status=yellow&timeout=1s"

# Request smaller persistent volumes.

volumeClaimTemplate:

accessModes: [ "ReadWriteOnce" ]

storageClassName: "standard"

resources:

requests:

storage: 15Gi

## Install and test

That's all of the configuration needed. Now you need to upgrade your Helm deployment with the newest configuration values. To upgrade your Helm deployment, run the following command:

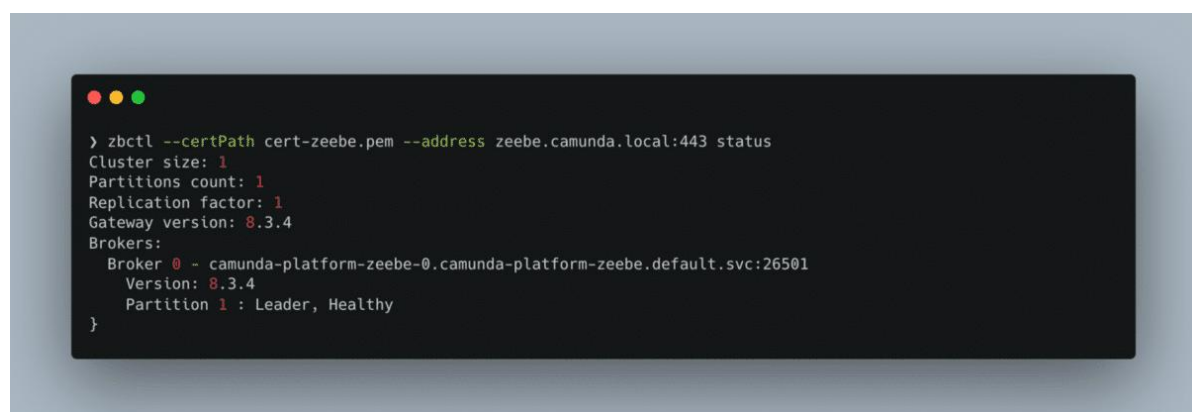
```
$ helm upgrade --install camunda-platform camunda/camunda-platform -f kind-combined-ingress.yaml
```

That's it! Now it's time to test! The first thing you can do is open <https://camunda.local/operate> or <https://camunda.local/tasklist> to make sure those applications open. Because we used a self-signed certificate, your browser may give a warning about not being able to verify the certificate. That is expected, you can click through the warning to get to the site. If you use a CA-signed certificate you will not see a warning.

The last thing to test is the gRPC connection to Zeebe Gateway. There are different ways to test this, but for this exercise I am going to use the [zbctl command line utility](#). Follow the instructions in the documentation to install it, then run the following command:

```
$ zbctl status --certPath cert-zeebe.pem --address zeebe.camunda.local:443
```

We are providing the self-signed certificate to zbctl because without it, zbctl wouldn't be able to validate the certificate and would fail with a warning similar to what you saw in your browser. We are also providing the address and port that we configured for the ingress, and the ingress controller is automatically routing that port to the gRPC port 26500 internally. If everything is set up correctly, you should see something similar to this:

A terminal window with a dark background and light green text. The command executed is 'zbctl --certPath cert-zeebe.pem --address zeebe.camunda.local:443 status'. The output shows cluster details: Cluster size: 1, Partitions count: 1, Replication factor: 1, Gateway version: 8.3.4. It also lists a broker: Broker 0 - camunda-platform-zeebe-0.camunda-platform-zeebe.default.svc:26501, Version: 8.3.4, and its partition: Partition 1 : Leader, Healthy.

```
> zbctl --certPath cert-zeebe.pem --address zeebe.camunda.local:443 status
Cluster size: 1
Partitions count: 1
Replication factor: 1
Gateway version: 8.3.4
Brokers:
  Broker 0 - camunda-platform-zeebe-0.camunda-platform-zeebe.default.svc:26501
    Version: 8.3.4
    Partition 1 : Leader, Healthy
}
```

## What's Next?

*Congratulations!* 🎉 Not only do you have Camunda Self-Managed running locally, it is now secured behind a certificate with a working ingress!

Here are some ideas for what to challenge yourself with next:

- Add Identity and Optimize, configure the ingress, and test the authentication with `zbctl`
- Enable Inbound Connectors
- Deploy to a cloud provider such as AWS, GCP, OpenShift, or Azure

Challenge yourself!