Securing K8s Ingress Traffic with HashiCorp Vault PKIaaS and JetStack Cert-Manager

N

Nicolas Ehrman
Apr 7, 2020 · 9 min read ★

Introduction

It is no longer a secret for anyone, Security is a major issue for all companies and of course the management of TLS certificates is one of these issues.

However, certificate requests are rarely automated and still require contacting the team in charge of the PKI to generate them. Usually, this means the certificate is issued with a very long validity period as you don't want to ask for a new certificate every day.

The problem is that this is no longer sustainable in a Cloud environment which is by definition extremely volatile and with a zero-trust network, even more for containerized environments.

This is why we often use certificates with a wildcard that we install on our Kubernetes platform, but that does not solve the problem, it only hides it.

In this article, we will see how to automate the creation and management of the lifecycle of TLS certificates in a Kubernetes environment with HashiCorp Vault and its PKI secret engine as well as JetStack cert-manager.

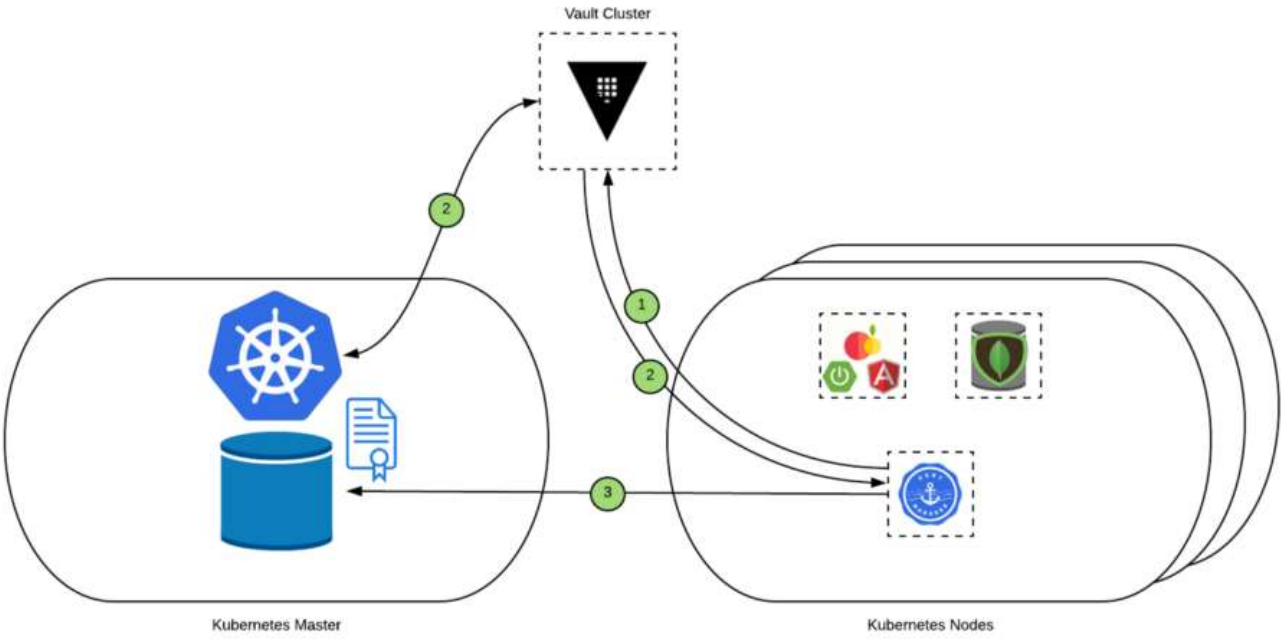
For the sake of simplicity and to make it repeatable for demos, almost everything is automated through HashiCorp Terraform, whether it be the deployment and configuration of JetStack Cert-Manager (using helm and Kubernetes provider) but also HashiCorp Vault (using Vault provider).

What do we want to achieve ?

As explained is the introduction, we want to build a common and automated workflow to allow creation and lifecycle management of TLS Certificates in a kubernetes environement for all Applications that are exposed to the outside world.

To do so, on one hand, we will deploy a Vault Server which will act as Root PKI and Intermediate PKI and will provide API endpoint for issuing Certificates; on the other hand, we will deploy and configure JetStack Cert-Manager which is a Certificate Controller Manager and it will be integrated with Vault.

In terms of workflow, it can be describe as follow:



1. Cert-Manager checks if a change occurs at the certificate object level and will use the information provided and sends a request through the issuer to Vault, as Vault

is supporting our Root and Intermediate PKI.

2. Before giving back a signed certificate, Vault will validate the identity and permissions of Cert-Manager issuer through Kubernetes authentication method. As soon as identity is validated, Vault will create and deliver a specific certificate signed by the intermediate CA which is valid for 10 minutes.

3. Finally, Cert-Manager will store the certificate as a Kubernetes secrets in ETCD and will renew as needed in regards of the lifetime of the certificate.

How to prepare your environment for this demo?

Here's what you need as prerequisites on your laptop (even if this demo could be done on any managed K8s cluster).

> ***Note**: For this demo, I will use a Java Application built by my dear friend Laurent Broudoux named Fruits-Catalog that I have forked for my own use.*

1. **Clone Repos**

First, even if it's quite obvious, you need to clone the repos needed by this demo.

- Clone Terraform Code repo:

```
$ git clone https://github.com/nehrman/medium-kubernetes-pkiaas
```

- Clone also the repo for the Java app:

```
$ git clone https://github.com/nehrman/secured-fruits-catalog-k8s
```

2. **Install & Configure Minikube**

> ***Note**: This is optionnal as you can do it on your existing Kubernetes environment.*

You need to install and configure Minikube on your laptop.

- To do so, on mac os, you have to launch this command to install minikube:

```
$ brew cask install minikube
```

- Then, you have to enable **Nginx** Ingress Router:

```
$ minikube addons enable ingress
```

- Start Minikube with your driver of choice (I am using VMware driver) and with enough resources to handle all services we need:

```
$ minikube start — vm-driver vmware — memory 8096 — cpus 4 — disk-size 50GB
```

You should end up with something like this:

- Now, we have to install Helm (It's easier and more secure since version 3):

```
$ brew install helm
```

3. **Start a Vault Server in Dev Mode**

In order to use PKI Secret engine from HashiCorp Vault, you need to start a Vault server on your local machine.

- Download Vault from the HashiCorp website, unzip it and add it to your path:

```
$ curl https://releases.hashicorp.com/vault/1.3.4/vault_1.3.4_darwin_amd64.zip -o vault.zip
```

```
$ unzip vault.zip
$ sudo mv vault /usr/local/bin
```

- For the sake of simplicity, we're using a vault server in dev mode, that means everything will be stored in memory. Of course, you can decide to use your existing vault or to build a real one using our reference architecture.

> ***Note**: HashiCorp Vault server in Dev mode is **not for production** but for development only.*

```
$ vault server -dev -dev-listen-address="0.0.0.0:8200" -dev-root-
token-id="root"
```

Now, all prerequisites are met and we can move forward …

How to deploy the Demo?

Ok, now, we're ready to use Terraform and our code to deploy and configure everything we need to test and validate the automation of our certificate management.

> *Note: This article will not cover how to download and configure Terraform, as we supposed you already have it.*

But first things first, even if everything is fully automated, it still worth it to understand what the code will do.

On Kubernetes:

- Create two namespaces, 1 for Cert-Manager and 1 for the application
- Create Vault Service Account and the clusterRoleBinding allowing vault to validate the identity of the Pod through Kubernetes API
- Deploy and configure Cert-Manager using Helm
- Create a Cert-Manager service account in the application namespace allowing the certificate issuer to authenticate through Vault
- Configure ingress route for our App
- Deploy MongoDB on K8s

- Create the Certificate object related to the application inside the Application namespace

On Vault:

- Configure Kubernetes Authentication Method to allow Cert-Manager to authenticate using a service account
- Configure Vault PKI Secret Engine for Root certificate
- Configure Vault PKI Secret Engine for Intermediate certificate
- Configure a Role to be able to generate certificate on-demand
- Configure a policy called “fruits-catalog” that allow access to PKI to be able sign request and issue certificate

Don’t worry, it is not as complicated as it looks like. :)

Now, let’s move on and configure our variables and run terraform to apply it.

Configure Kubernetes, HashiCorp Vault and JetStack Cert-Manager

Ok, let’s start with the deployment and configuration of all the components we need for our demo.

1. **Configure Env variables**

To be able to connect to vault, we need to set up VAULT_TOKEN with this command:

```
$ export VAULT_TOKEN=root
```

That’s the only environment variable we have to configure for Vault’s provider for Terraform.

If you want to customize the domain name or any other parameters, all others variables are defined in the **variables.tf** file and can be changed as needed.

> ***Note: Don’t forget to configure the Vault’s address in your variable file or Terraform will not be able to contact Vault Server.***

2. It's time to use Terraform

That's not the final step, but at least, you don't have to do everything manually.

- Run terraform init to prepare the environment:

```
$ terraform init
```

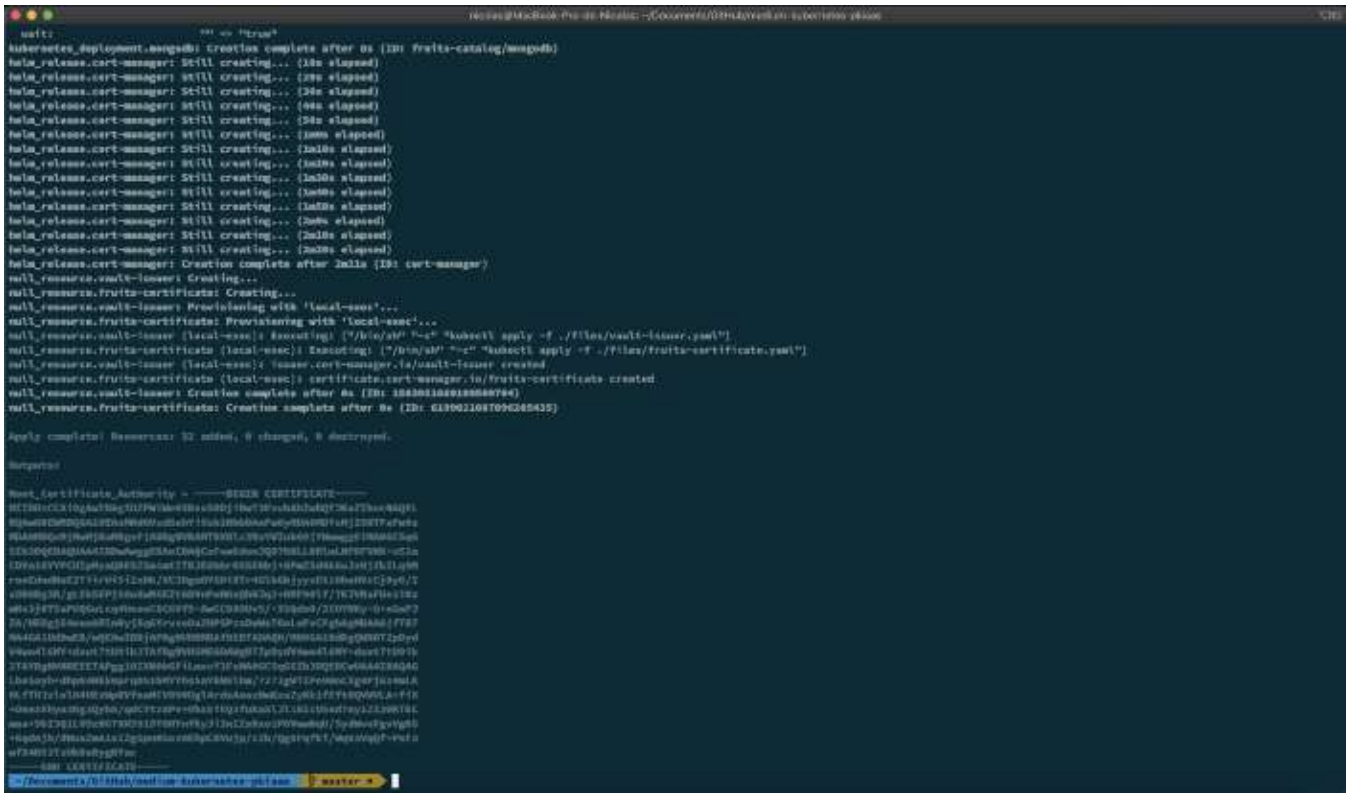
- Run terraform plan to check if everything is correct:

```
$ terraform plan
```

- And finally, run terraform apply to make the magic happen:

```
$ terraform apply
```

You should end up with something like this:



> ***Note**: Store the Root CA somewhere as you'll need it later on for your web browser.*

You can also check the state of all the newly created Kubernetes objects by running these commands:

```
$ kubetctl get sa -n default
NAME       SECRETS   AGE
default    1         134m
vault-sa   1         10m
$ kubectl get pods -n cert-manager
NAME                                READY   STATUS    RESTARTS   AGE
cert-manager-75f6cdcb64-9jcbb       1/1     Running   0          8m25s
cert-manager-cainject...            1/1     Running   0          8m25s
cert-manager-webhook-...            1/1     Running   0          8m25s
$ kubectl get sa -n fruits-catalog
NAME              SECRETS   AGE
cert-manager-sa   1         12m
default           1         12m
$ kubectl get issuer -n fruits-catalog
NAME           READY   AGE
vault-issuer   True    11m
$ kubectl get certificate -n fruits-catalog
NAME                 READY   SECRET               AGE
fruits-certificate    True    fruits-certificate   13m
```

Then, you can check with more details the vault-issuer and fruits-catalog certificate with these commands :

```
$ kubectl describe issuer vault-issuer -n fruits-catalogs
$ kubectl describe certificate fruits-catalogue -n fruits-catalogs
```

We are almost done as everything is configured but we still don't have an Application to show something nice and to test our TLS Certificate.

Finally, you can deploy the fruits-catalog app using Maven

As we are almost done, let's do the last steps in order to have a fully configured Application with a TLS Ingress route configured and validated by our PKIaaS from HashiCorp Vault.

1. **Prepare your environment**

As we use minikube and our local Docker environment, we have to configure our bash environment for using the docker daemon inside minikube:

```
$ eval $(minikube docker-env)
```

2. Use maven to build and deploy the fruits-catalog app

It's time to deploy the app and make sure everything works as expected.

- Run maven to build the app (if maven isn't already installed on your laptop, run brew install maven):

```
$ mvn fabric8:deploy -Pkubernetes -Dfabric8.namespace=fruits-catalog
```

You should end up with something like this:

```
bric8/openshift.yml to /Users/nicolas/.m2/repository/com/github/lbroudoux/msa/fruits-catalog/1.0.0-SNAP
SHOT/fruits-catalog-1.0.0-SNAPSHOT-openshift.yml
[INFO]
[INFO] <<< fabric8-maven-plugin:4.3.1:deploy (default-cli) < install @ fruits-catalog <<<
[INFO]
[INFO]
[INFO] --- fabric8-maven-plugin:4.3.1:deploy (default-cli) @ fruits-catalog ---
[INFO] F8: Using Kubernetes at https://192.168.94.135:8443/ in namespace fruits-catalog with manifest /
Users/nicolas/Documents/GitHub/secured-fruits-catalog-k8s/target/classes/META-INF/fabric8/kubernetes.ym
l
[INFO] F8: Using namespace: fruits-catalog
[INFO] F8: Using namespace: fruits-catalog
[INFO] F8: Creating a Service from kubernetes.yml namespace fruits-catalog name fruits-catalog
[INFO] F8: Created Service: target/fabric8/applyJson/fruits-catalog/service-fruits-catalog-49.json
[INFO] F8: Using namespace: fruits-catalog
[INFO] F8: Creating a Deployment from kubernetes.yml namespace fruits-catalog name fruits-catalog
[INFO] F8: Created Deployment: target/fabric8/applyJson/fruits-catalog/deployment-fruits-catalog-50.jso
n
[INFO] F8: HINT: Use the command `kubectl get pods -w` to watch your pods start up
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  30.277 s
[INFO] Finished at: 2020-04-06T12:21:26+02:00
[INFO] ------------------------------------------------------------------------
```

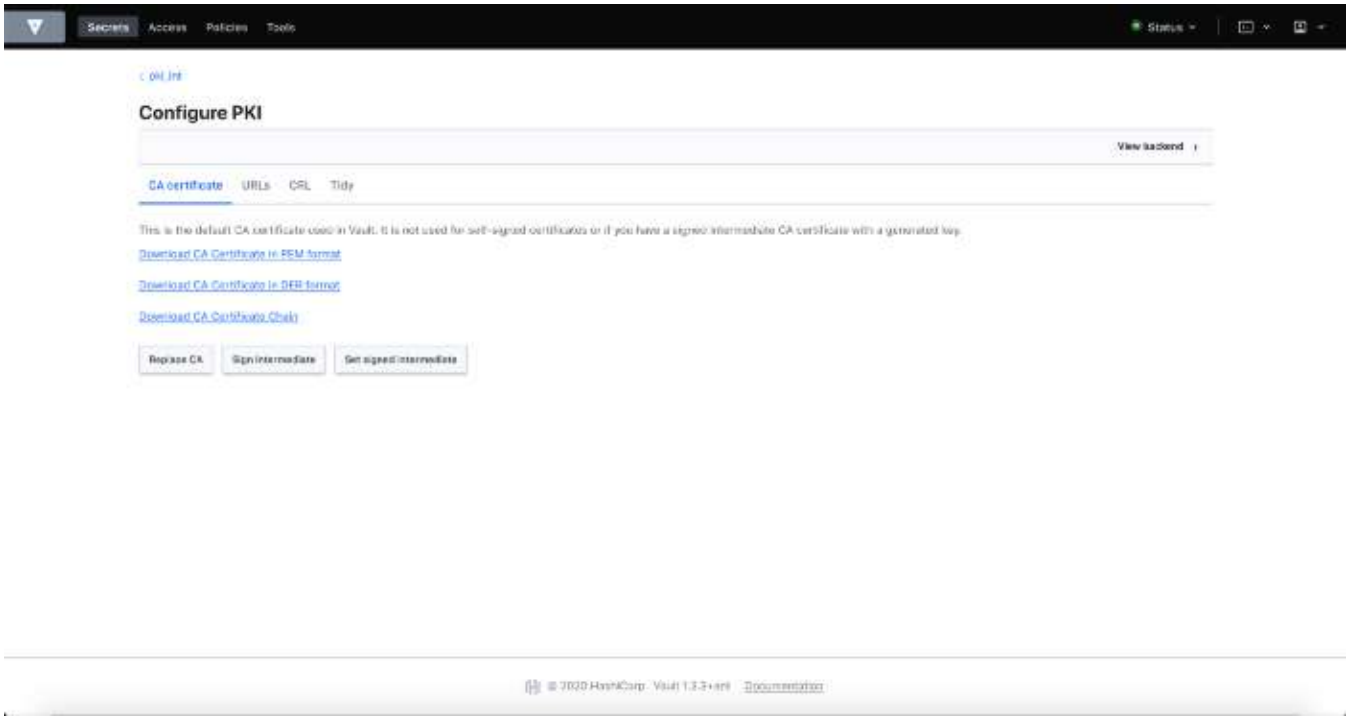
Check also that you have a new pod in the fruits-catalog namespace :

```
$ kubectl get pods -n fruits-catalog
NAME                              READY   STATUS    RESTARTS   AGE
fruits-catalog-6b45f4554d-k6wkp   1/1     Running   0          74s
mongodb-58845f7854-f2rpx          1/1     Running   0          40m
```

3. It's time to test if the certificate is well generated and configured and if everything works

Now, let's retrieve the CA Certificate Chain and configure our laptop to trust the certificate and test https connection to the app.

- Retrieve the certificate from Vault UI and import it on your favorite browser. To do so, via Vault WebUI, click on "*PKI_INT*", then "*Configuration*" and "*Configure*".

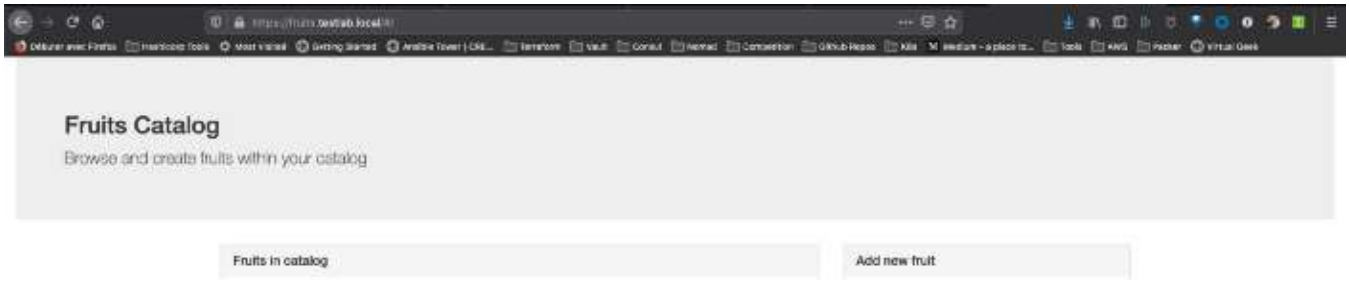


As we use Vault as Root and Intermediate PKI, you have to add the CA Chain to your favorite Web Browser to allow it to trust Vault.

Of course, il a real life scenario, the ROOT PKI of the company will be used to sign Vault as an intermediate authority.

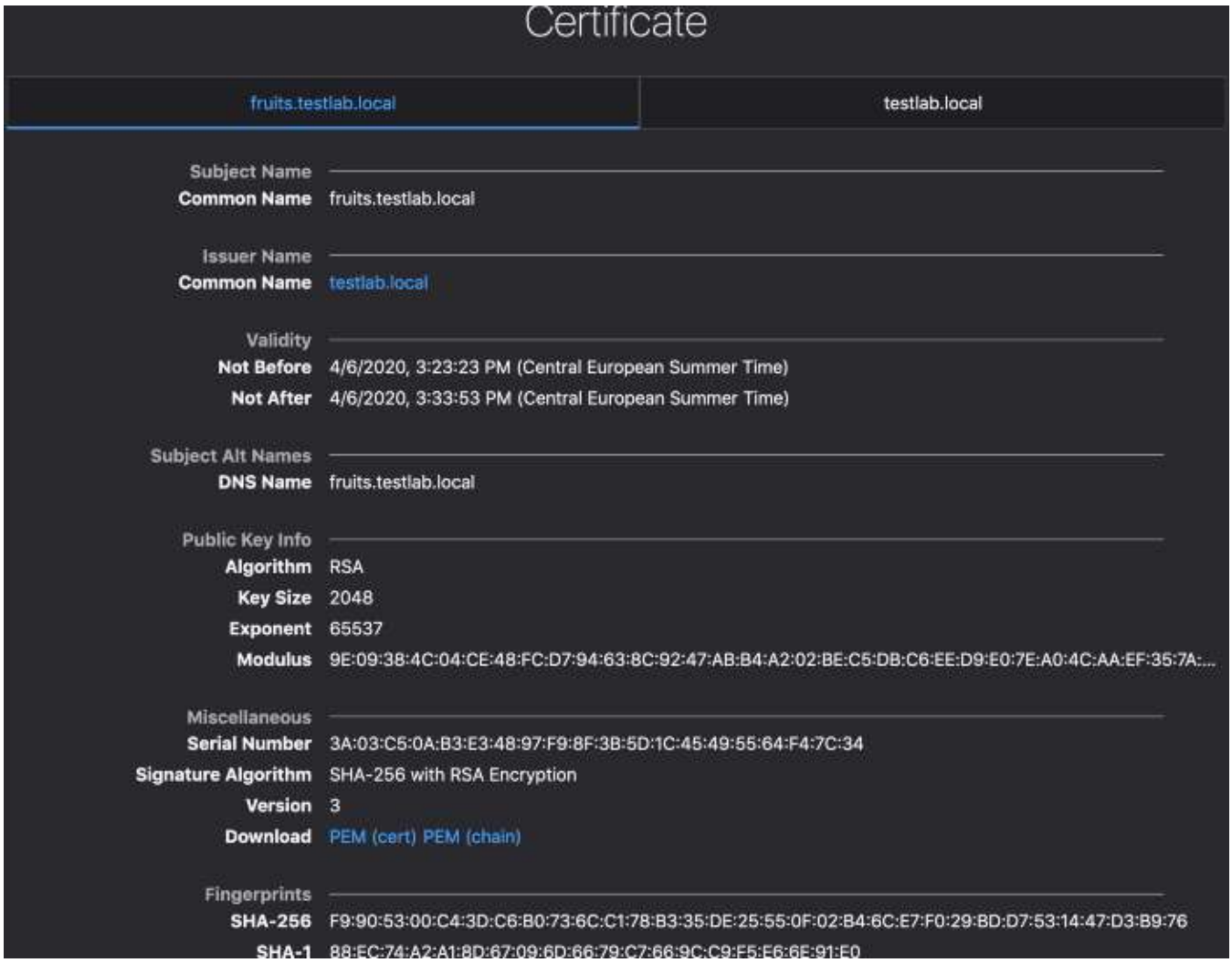
- Connect with your favorite browser to **https://fruits.testlab.local**

> ***Tips****: If it's not already done, create an entry in your host with the minikube's IP pointing to fruits.testlab.local*



Look at the certifcate an ensure it is correctly issued by HashiCorp Vault and your own PKI that we've configured in the previous steps:

Let's take a closer at the certificate:



Have you seen that the certificate is only valide for 10 Minutes ? But wait, your application is still working with a valid certificate.

Yes, you're right, HashiCorp Vault and JetStack Cert-manager can issue certificates but they will also manage their lifecycle. This means Cert-Manager is aware of the lifetime of the certificate delivered by Vault and it will request another one (before the end of life of the first one) automatically to ensure continuous operation of the Application while providing better security with short time lived Certificates.

Conclusion

In this article, we tackle a challenge that most of company are facing and we demonstrate how to make the PKI as automated as possible. And, yes, we used Kubernetes to demonstrate that, but, we can do more by automatically delivering certificates to Load Balancer, Web Server (VMs) or anything else that needs a TLS certificate with exactly the same workflow.

We can go even further with HashiCorp Vault Enterprise by providing, for instance, namespaces where you can provide multi-tenancy to your internal or external

customers.

Thank you for your time reading this article, it's not the easiest subject but it gives you a glance at the power of HashiCorp Tools coupled with JetStack Cert-Manager for managing TLS Certificates.

References:

- Vault Kubernetes Auth Method
- Vault PKI Secret Engine
- JetStack Cert-Manager with Vault
- Terrafrom Helm Provider
- Terraform Kubernetes Provider
- Terraform Vault Provider

Vault Hashicorp K8s Pki Certificate