K8S CON OPA GATEKEEPER

Daniel Pérez Gallo
2ºASIR

ÍNDICE

1. Introducción

El siguiente documento es el proyecto final del ciclo formativo de grado superior de administración de sistemas informáticos en red. En el mismo se va a detallar la información sobre un gestor de políticas diseñado para Kubernetes (K8S), el cual se llama Open Policy Agent y utiliza un plugin llamado Gatekeeper.
Se explicará con detalle para que sirve, como funciona y se indicará una prueba para comprobar su funcionamiento e instalación.

1.1. Objetivos que se han conseguido:

Kubernetes nos ofrece gran cantidad de posibilidades y esto nos a permitido:

- Aprender a crear políticas con OPA (Open Policy Agent).

- Aprender a utilizar Gatekeeper, tanto como instalarlo y crear diferentes políticas a partir de su lenguaje Rego.

2. ¿Qúe es kubernetes?

Kubernetes es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios. Kubernetes facilita la automatización y la configuración declarativa. Tiene un ecosistema grande y en rápido crecimiento. El soporte, las herramientas y los servicios para Kubernetes están ampliamente disponibles.

Kubernetes tiene varias características y puede ser:
- Una plataforma de contenedores
- Una plataforma de microservicios
- Una plataforma portable de nube

Ofrece un entorno de administración centrado en contenedores. Kubernetes orquesta la infraestructura de cómputo, redes y almacenamiento para que las cargas de trabajo de los usuarios no tengan que hacerlo.

2.1. ¿Por qué usar contenedores?

Estos se despliegan basados en virtualización a nivel del sistema operativo, en vez del hardware. Estos contenedores están aislados entre ellos y con el servidor anfitrión: tienen sus propios sistemas de archivos, no ven los procesos de los demás y el uso de recursos puede ser limitado. Son más fáciles de construir que una máquina virtual, y porque no están acoplados a la infraestructura y sistema de archivos del anfitrión.

Ya que los contenedores son pequeños y rápidos, una aplicación puede ser empaquetada en una imagen de contenedor. Con contenedores, podemos crear imágenes inmutables al momento de la compilación en vez del despliegue, esto permite tener un entorno consistente que va desde desarrollo hasta producción. De igual forma, los contenedores son más transparentes que las máquinas virtuales y eso hace que el monitoreo y la administración sean más fáciles.

3. Políticas

3.1.¿Qué es una política?

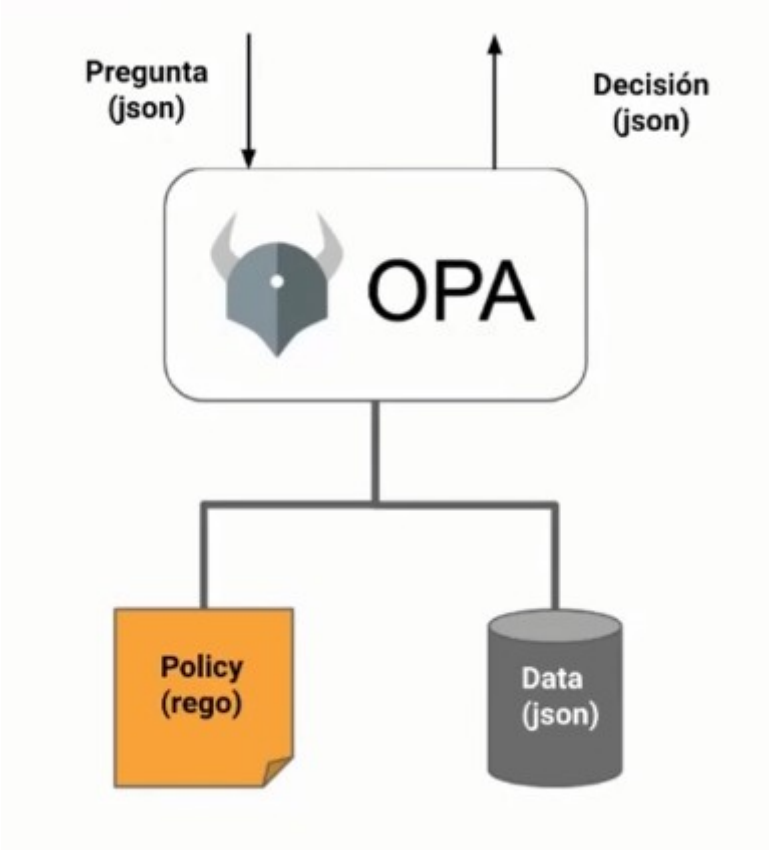
Una política es un conjunto de reglas que su autor declara para que un sistema cumpla con un conjunto determinado de requisitos. Estas se escriben con el lenguaje anteriormente mencionado, Rego.

Esta sería una plantilla de ejemplo de declaración para las políticas:

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8srequiredlabels
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredLabels
        listKind: K8sRequiredLabelsList
        plural: k8srequiredlabels singular: k8srequiredlabels
     validation: # Schema for the `parameters` field openAPIV3
     Schema:
     properties:
          labels:
          type: array items: string targets:
          - target: admission.k8s.gatekeeper.sh
          rego: |
            package k8srequiredlabels
            deny[{"msg": msg, "details": {"missing_labels":
            missing}}] { provided := {label |
            input.review.object.metadata.labels[label]} required :=
            {label | label := input.parameters.labels[_]}
            missing := required - provided count(missing) > 0
            msg := sprintf("you must provide labels: %v",
            [missing]) }
```

3.2. ¿Qué es un gestor de políticas (Open Policy Agent)?

Es una herramienta que intenta desacoplar la política y el forzado de la misma, al contrario de lo que se suele ver frecuentemente, ya que política y el forzado suelen ir de la mano.



OPA utiliza un lenguaje declarativo exclusivo para él llamado Rego.

Su funcionamiento es muy sencillo, realizamos una pregunta mediante un fichero .json, y este nos devuelve la política en forma de fichero .json, con un sí o no.

OPA posee gran variedad de plugins y todos cubren todo lo relacionado con proyectos open source.

4. RBAC (Role Based Access Control)

Es un tipo de gestión de identidades y acceso que involucra una serie de permisos o plantillas que determinan quién (sujeto) puede realizar qué (verbo) y en qué parte (recurso de la API de Kubernetes).
Esto es la evolución del control de acceso basado en atributos (ABAC) tradicional, el cual permite el acceso en función del nombre del usuario y no de sus responsabilidades.

4.1. ¿Qué son las funciones?

Las funciones conceden distintos niveles de acceso a los pods y los nodos, y pueden tener autorización para acceder a un grupo específico de clústeres que trabajan juntos en forma de carga de trabajo de una aplicación (funciones) o a clústeres completos (funciones del clúster).

- Las funciones permiten acceder a los grupos de clústeres vinculados virtualmente que se conocen como espacios de nombres. Dichas funciones son recursos con espacios de nombres. El enlace de funciones se utiliza para consolidar los usuarios, los grupos de usuarios o los nombres de las cuentas de servicio en una sola función.
- Las funciones del clúster, las cuales abarcan varios espacios de nombres, permiten acceder a clústeres completos, que son grupos de nodos de hardware individuales. Los enlaces de funciones del clúster asocian una función a cada espacio de nombres.

Es posible usar los metadatos para combinar y apilar los enlaces y los permisos de las funciones del clúster. Esto concede las autorizaciones definidas en una función del clúster a los recursos que se encuentran en el espacio de nombres del enlace de funciones, lo cual ayuda a definir las que son comunes a todo un clúster, que pueden reutilizarse en varios espacios de nombres.

4.2. ¿Cómo funciona el RBAC de Kubernetes?

La API de Kubernetes es el frontend del plano de control de este sistema, la cual comunica las interacciones con una computadora o un sistema con el fin de recuperar información o realizar una función.

El RBAC de Kubernetes recopila las solicitudes de funciones relacionadas en grupos de API, que se comunican con los servidores de API cuando se conectan ciertas funciones a los extremos de la misma.

4.3. ¿Qué no puede hacer RBAC?

No puede comprobar todas las etiquetas (labels) que tiene un pod, es decir, nos permite crear un pod, pero le resulta indiferente lo que haya dentro del mismo o de cualquier otro objeto.

5. Gatekeeper

5.1.¿Qué es Gatekeeper?

Gatekeeper es uno de esos plugins de OPA que hemos mencionado en el apartado 3.

Se caracteriza por forzar una decisión que tomamos para un entorno de kubernetes.

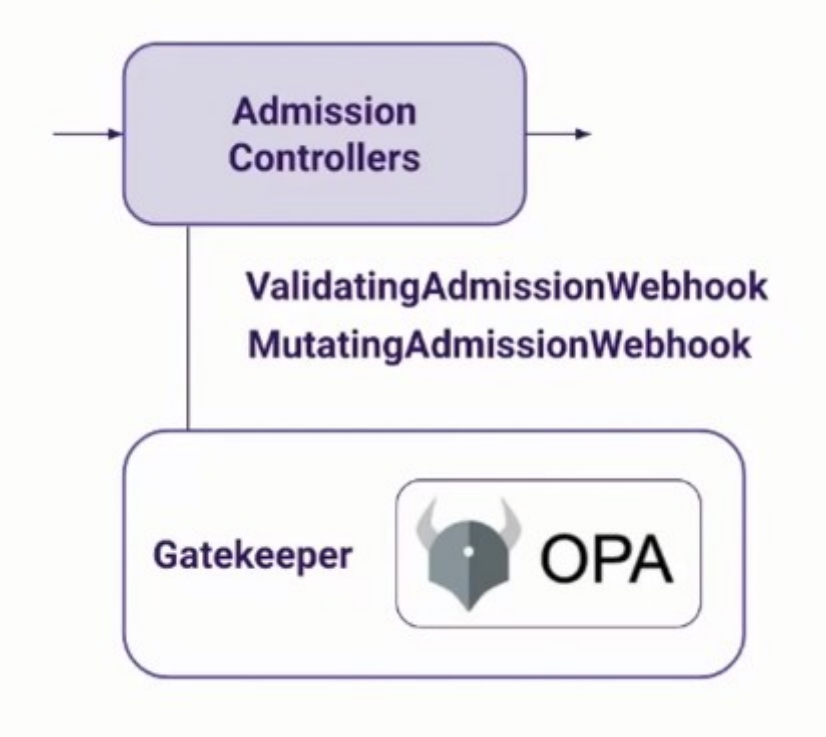
5.2. ¿Cómo funciona Gatekeeper?

Gatekeeper utiliza un webhook mediante una API que posee Kubernetes, este verifica si las políticas se cumplen o no para cargar dicha solicitud. También nos proporciona un lenguaje llamado Rego, el cual nos permite especificar que políticas queremos aplicar.

La forma que tiene Gatekeeper de forzar las políticas es la siguiente:

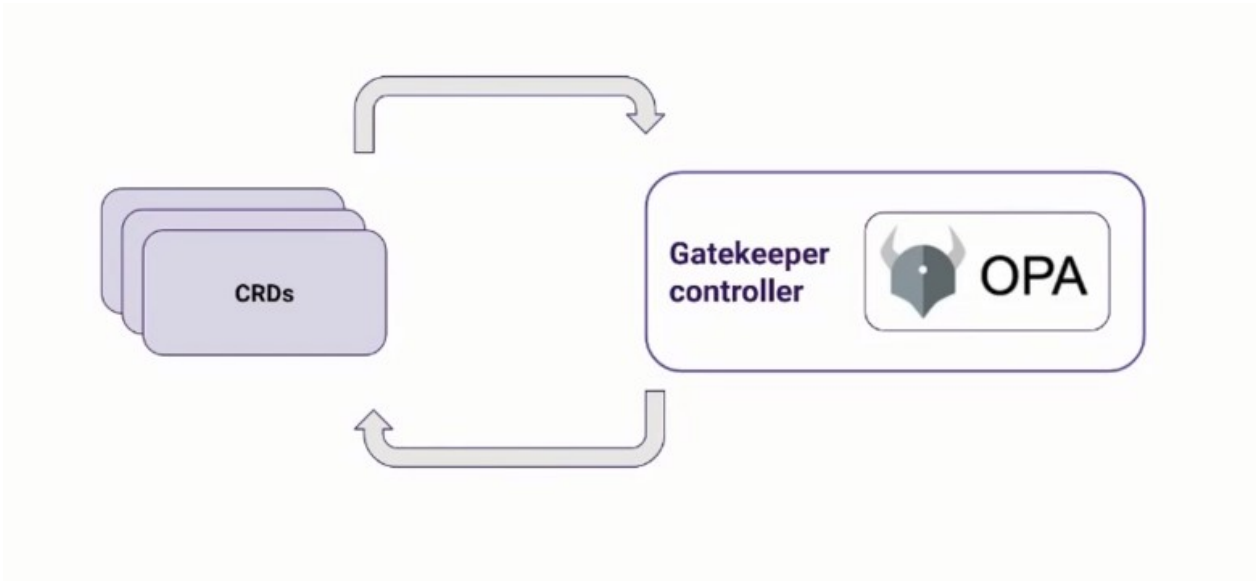
La autenticación seríamos nosotros, mientras que la autorización sería RBAC.

Mientras que los Controles de acceso son una serie de reglas específicas que están en el binario del servidor de la API de kubernetes, nosotros podemos habilitarlos o desactivarlos cuando arrancamos el servidor de kubernetes.

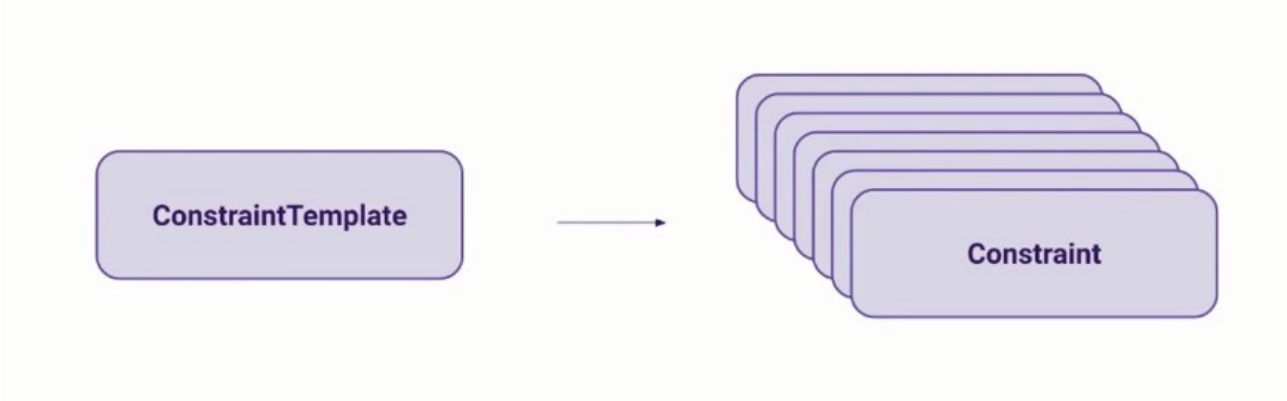


Tenemos dos binarios bastante importantes y son específicos del mismo, ValidatingAdmissionWebhook y MutatingAdmissionWebhook, que sin tener que cambiar el código de la API de kubernetes nos permite introducir nuestro propio código.

Gatekeeper lo que hace es un ValidatingAdmissionWebhook que va a hacer que cuando una request está pasando por los pasos mencionados previamente, concretamente en el AdmissionControllers va a comprobar que cumple con las políticas de nuestro clúster.



Gatekeeper es un plugin primitivo, completamente diseñado para kubernetes, ya que crea nuevos objetos para la API de kubernetes mediante CRDs y un controller que hace el “Reconciliation Loop” (Esto sirve para conducir el estado actual hacia el estado deseado).



De los CRDs que se crean, los principales son el ConstraintTemplate y los Constraint.

En el primero es donde vamos a definir las políticas a nivel genérico usando el lenguaje Rego. A partir de dichas políticas vamos a poder hacer distintas instancias para cosas particulares de nuestro clúster.

5.3. Reutilización de políticas

Gatekeeper facilita mucho la reutilización de las políticas, esto debido a varios factores:

- Las imágenes proceden únicamente de repositorios conocidos

- Los deployment tienen que tener una serie de etiquetas

- Las imágenes requieren un digest (identificador inmutable para la imágen de un contenedor)

- Los contenedores tienen que establecer límites para CPU y memoria dentro de unos valores

5.4. Gatekeeper Library

En GitHub existe un repositorio/librería open source llamado Gatekeeper Library, el cuál se puede reutilizar para muchas tareas comunes.

Enlace al repositorio:
https://github.com/open-policy-agent/gatekeeper-library/tree/master/library/general

Por ejemplo si quisieramos que nuestros ingress fueran exclusivamente https y no http, deberíamos recurrir a la plantilla “httpsonly” del repositorio, la cual se ubica en la siguiente url:

https://github.com/open-policy-agent/gatekeeper-library/blob/master/artifacthub/library/general/httpsonly/1.0.0/template.yaml

Este repositorio no solo nos proporciona la plantilla y el código en rego, sino que también nos proporciona varios ejemplos de como serían unas instancias mostrándonos que tipo de objetos se permiten y otra la cuál va a fallar dicha política.

Esta sería la permitida:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-demo-allowed
  annotations:
    kubernetes.io/ingress.allow-http: "false"
spec:
  tls: [{}]
  rules:
    - host: example-host.example.com
      http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
           service:
              name: nginx
              port:
                number: 80
```

La siguiente fallaría:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-demo-disallowed
spec:
  rules:
    - host: example-host.example.com
      http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: nginx
              port:
                number: 80
```

6. Rego

Anteriormente hemos mencionado un lenguaje llamado Rego, este es un lenguaje de políticas declarativo creado específicamente para gatekeeper.

Esta creado para expresar políticas sobre estructuras jerárquicas complejas de datos.

Esto sería un ConstraintTemplate de ejemplo:

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
        name: k8srequiredlabels
spec:
        crd:
                spec:
                        names:
                                kind: k8srequiredLabels
                        validation:
                        #Schema for the 'parameters' field
                        openAPIV3Schema:
                                properties:
                                        labels:
                                                type: array
                                                items: string
                targets:
                - target: admission.k8s.gatekeeper.sh
                        rego: |
                                package k8srequiredlabels

                                violation[{"msg": msg, "details":
{"missing_labels": missing}}] {
                        provided := {label |
input.review.object.metadata.labels[label]}
                        required := {label | label :=
input.parameters.labels[_]}
                        missing:= required - provided
                        count(missing) > 0
                        msg := sprintf("you must provide
labels: %v", [missing]}}
```

Constraints de ejemplo:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: k8sRequiredLabels
metadata:
     name: ns-must-have-gk
spec:
     match:
          kinds:
               - apiGroups: [“”]
                    kinds: [“Namespace”]
     parameters:
          labels: [“gatekeeper”]
```

En este constraint estamos indicando que todos los Namespaces de nuestro clúster tienen que tener el label “gatekeeper”.

Otro ejemplo pero esta vez para un pod:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: k8sRequiredLabels
metadata:
     name: pod-required-labels
spec:
     match:
          namespace: “default”
          kinds:
               - apiGroups: [“”]
                    kinds: [“Pod”]
     parameters:
          labels: [“do-not-delete”]
```

Estamos indicando que todos los pods tienen que tener el label “do-not-delete”

7. Instalación

Antes de todo nos aseguraremos que tengamos permisos admin de RBAC:

```
kubectl create clusterrolebinding cluster-admin-
binding \
--clusterrole cluster-admin \
--user <YOUR USER NAME>
```

Para instalar gatekeeper necesitamos un fichero gatekeeper.yaml, el cual se puede obtener de la siguiente url:

https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.3/deploy/gatekeeper.yaml

Lo bajamos en nuestra máquina:

```
$ kubectl apply -f https://raw.githubusercontent.com/
open-policy-agent/gatekeeper/master/deploy/
gatekeeper.yaml
```

Realizamos una comprobación de que se ha instalado correctamente observando los namespaces:

```
$ kubectl get ns
NAME                   STATUS   AGE
default                Active   2d17h
gatekeeper-system      Active   47s
kube-node-lease        Active   2d17h
kube-public            Active   2d17h
kube-system            Active   2d17h
kubernetes-dashboard   Active   2d17h
```

Como podemos observar se ha creado un namesapace llamado “gatekeeper-system”

En cuanto a los pods:

```
$ kubectl get pods -n gatekeeper-system
```

```
NAME                                             READY   STATUS    RESTARTS   AGE
gatekeeper-audit-6b54b46d76-chsc9                1/1     Running   0          31s
gatekeeper-controller-manager-66458f487-g8mbh    1/1     Running   0          31s
gatekeeper-controller-manager-66458f487-g8mbh    1/1     Running   0          31s
gatekeeper-controller-manager-66458f487-g8mbh    1/1     Running   0          31s
```

Y por último los cdrs que se han creado son los siguientes:

```
$ kubectl get crd
```

```
NAME                                                  CREATED AT
configs.config.gatekeeper.sh                          2022-12-14T17:01:42Z
constraintpodstatuses.status.gatekeeper.sh            2022-12-14T17:01:42Z
constrainttemplatepodstatuses.status.gatekeeper.sh    2022-12-14T17:01:42Z
constrainttemplates.templates.gatekeeper.sh           2022-12-14T17:01:42Z
```

8. Pruebas prácticas

Para la realización de estas pruebas se va a utilizar el repositorio open source anteriormente mencionado.

Vamos a utilizar el código de la librería requieredlabels, al cual podemos acceder desde el siguiente enlace:
https://github.com/open-policy-agent/gatekeeper-library/blob/master/artifacthub/library/general/requiredlabels/1.0.0/template.yaml

Creamos dicho template en nuestra máquina con el siguiente comando:

```
$ kubectl apply -f template.yaml
```

Esto nos va a crear un CRD que podremos ver con el comando siguiente:

```
$ kubectl get crd
NAME                                                  CREATED AT
configs.config.gatekeeper.sh                          2022-12-14T17:01:42Z
constraintpodstatuses.status.gatekeeper.sh            2022-12-14T17:01:42Z
constrainttemplatepodstatuses.status.gatekeeper.sh    2022-12-14T17:01:42Z
constrainttemplates.templates.gatekeeper.sh           2022-12-14T17:01:42Z
k8srequiredlabels.constraints.gatekeeper.sh           2022-12-14T18:08:46Z
```

A continuación vamos a hacer una regla para nuestro cluster, para ello nos dirigimos a la carpeta samples ubicada en el mismo directorio del repositorio donde tenemos el fichero que acabamos de ver, ahí hallaremos un fichero constraint.yaml con el siguiente contenido:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: all-must-have-owner
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Namespace"]
  parameters:
    message: "All namespaces must have an `owner` label that
points to your company username"
    labels:
      - key: owner
        allowedRegex: "^[a-zA-Z]+.agilebank.demo$"
```

Modificaremos dicho fichero para que falle, quedaría tal que así:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: prueba-fallo
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
  parameters:
    message: "All pods must have an `prueba-fallo` label that
points to your company username"
    labels:
      - key: prueba-fallo
    allowedRegex: "^[a-zA-Z]+.agilebank.demo$"
```

Aplicamos los cambios:

```
$ kubectl -f apply constraint.yaml
```

Vamos a crear un pod que no tenga dicha label,

Por ejemplo definimos un pod con el siguiente contenido en un fichero nginx.yaml:

```
apiVersion: v1
kind: Pod
metadata:
creationTimestamp:  null
labels:
run: nginx
name: nginx
spec:
containers:
-image: nginx
 name: nginx
 resources: {}
dnsPolicy: ClusterFirst
restartPolicy: Always
status: {}
```

Al ejecutar esto se nos producirá el siguiente error:

```
$ kubectl apply -f nginx.yaml
Error from server ([denied by prueba-fallo] All
pods must have an ‘prueba-fallo’ label): error when
creating “nginx.yaml”: admission webhook
“validation.gatekeeper.sh” denied the request:
[denied by all-must-have-kcdspain] All pods must
have an ‘prueba-fallo’ label
```

Vamos a modificar el fichero nginx.yaml:

```
apiVersion: v1
kind: Pod
metadata:
creationTimestamp:  null
labels:
run: nginx
prueba-fallo: nginx
spec:
containers:
-image: nginx
 name: nginx
 resources: {}
dnsPolicy: ClusterFirst
restartPolicy: Always
status: {}
```

Y de esta forma lo aplicamos y vemos que se ha creado correctamente:

```
$ kubectl apply -f nginx.yaml
pod/nginx created
```

Vamos a obtener el objeto que hemos creado:

```
$ kubectl get K8sRequiredLabels
NAME            AGE
prueba-fallo    53S
```

El host tiene que ser único para todos los objetos Ingress para ello se crea un crd con el siguiente contenido (normalmente suele ser un template.yaml):

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8suniqueingresshost
  annotations:
    metadata.gatekeeper.sh/title: "Unique Ingress Host"
    metadata.gatekeeper.sh/version: 1.0.2
    metadata.gatekeeper.sh/requiresSyncData: |
      "[
        [
          {
            "groups": ["extensions"],
            "versions": ["v1beta1"],
            "kinds": ["Ingress"]
          },
          {
            "groups": ["networking.k8s.io"],
            "versions": ["v1beta1", "v1"],
            "kinds": ["Ingress"]
          }
        ]
      ]"
spec:
  crd:
    spec:
      names:
        kind: K8sUniqueIngressHost
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8suniqueingresshost
        identical(obj, review) {
          obj.metadata.namespace == review.object.metadata.namespace
          obj.metadata.name == review.object.metadata.name
        }
        violation[{"msg": msg}] {
          input.review.kind.kind == "Ingress"
          re_match("^(extensions|networking.k8s.io)$",
input.review.kind.group)
          host := input.review.object.spec.rules[_].host
          other := data.inventory.namespace[_][otherapiversion]["Ingress"]
[name]
          re_match("^(extensions|networking.k8s.io)/.+$", otherapiversion)
          other.spec.rules[_].host == host
          not identical(other, input.review)
          msg := sprintf("ingress host conflicts with an existing ingress <
%v>", [host])
        }
```

Lo aplicamos:

```
$ kubectl apply -f template.yaml
```

Y un fichero sync.yaml con lo siguiente:

```
apiVersion: config.gatekeeper.sh/v1alpha1
kind: Config
metadata:
  nombre: config
  namespace: "gatekeeper-system"
spec:
  sync:
    syncOnly:
      - group: "extensions"
        version: "v1beta1"
          kind: "Ingress"
       - group: "networking.k8s.io"
          version: "v1beta1"
          kind: "Ingress"
```

Lo aplicamos:

```
$ kubectl apply -f sync.yaml
config.config.gatekeeper.sh/config created
```

Vamos a realizar una comprobación de si esto funciona.
Para ello vamos a crear un constraint.yaml con el siguiente contenido:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sUniqueIngressHost
metadata:
  name: unique-ingress-host
spec:
  match:
    kinds:
      - apiGroups: ["extensions", "networking.k8s.io"]
        kinds: ["Ingress"]
```

Lo aplicamos:

```
$ kubectl apply -f constraint.yaml
```

Crearemos un fichero example_disallowed.yaml con este contenido:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
     name: ingress-host-disallowed
     namespace: default
spec:
    rules:
    - host: example-host.example.com
      http:
         paths:
         - pathType: Prefix
           path: "/"
           backend:
               service:
                  name: nginx
                port:
                    number: 80
```

Aplicamos:

```
$ kubectl apply -f example_disallowed.yaml
```

Como podemos comprobar se crea a la perfección y ahora crearemos un segundo, el cual no nos va a dejar crear, ya que existe el primero.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-host-disallowed2
  namespace: default
spec:
  rules:
  - host: example-host.example.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
       name: nginx
       port:
         number: 80
```

Lo creamos:

```
$ kubectl apply -f example_disallowed2.yaml
Error from server ([denied by unique-ingress-host] ingress host conflicts
with an existing ingress < example-host.example.com>
[denied by unique-ingress-host] ingress host conflicts with an existing
ingress <example-host.example.com): error when creating
“example_disallowed.yaml”: admission webhook “validation.gatekeeper.sh”
denied the request: [denied by unique-ingress-host] ingress host conflicts
with an existing ingress <example-host-example.com>
[denied by unique-ingress-host] ingress host conflicts with an existing
ingress <example-host.example.com>
```

9. Conclusiones y propuestas para seguir trabajando sobre el tema

En conclusión, Kubernetes y OPA Gatekeeper nos permite organizar nuestros contenedores mediante varias políticas que nosotros definimos, ya sea mediante las nuestras propias o las del repositorio de la comunidad open source.

Si deseamos seguir trabajando sobre este tema, la mejor sería empezar a aprender el lenguaje de Rego para el desarrollo de políticas de OPA Gatekeeper.

Rego es un lenguaje que de primeras puede resultar algo complicado, pero existen varios enfoques para abordar este tema.

Mi recomendación sería utilizar “The Rego Playground”.

Podemos acceder desde la siguiente url: https://play.openpolicyagent.org/

Esto es un servicio web que nos permite evaluar fácilmente nuestra política, podemos escribirlo en formato .json y nos lo traduce al lenguaje Rego.

Esta sería la mejor opción para seguir formándonos en este ámbito.

10. Bibliografía

Las siguientes páginas web es de donde se ha sacado principalmente la información sobre el tema para realizar la memoria.

Repositorio: https://github.com/open-policy-agent/gatekeeper

Kubernetes junto a Gatekeeper: https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/

Instalación, uso, funcionamiento…:
https://tanzu.vmware.com/developer/guides/gs-opa-gatekeeper/