Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2019/2020

Trabajo de Fin de Máster

Estudio de frameworks y técnicas serverless en Kubernetes

Autor: Òscar Illescas Jiménez
Tutor: Micael Gallego Carrillo

Tabla de contenido

Resumen

En este trabajo se intenta encontrar alternativas a los servicios en la nube de los proveedores de cloud en el ecosistema serverless.

Partiendo de microk8s como proveedor de Kubernetes, el estudio se va a centrar en instalar y probar algunas alternativas a los siguientes servicios.

- Funciones como servicio (FaaS)
- Gestión de ficheros
- Bases de datos
- Colas y gestión de eventos
- Gestión de usuarios

En la parte de FaaS, hemos probado 2 tecnologías de las muchas disponibles

- Knative
- OpenFaaS

OpenFaaS se ha estudiado más a fondo, ya que el desarrollo de funciones es la más similar de las dos a como se desarrollan las funciones en los proveedores en la nube.

En las bases de datos hemos probado un operador de bbdd relacionales y otro de no relacionales

- El operador de MongoDB de Percona.
- El operador de PostgreSQL de Zalando.

En cuanto a la gestión de ficheros hemos probado como alternativa a S3, Minio.

Para la gestión de usuarios se ha probado Keycloak de RedHat y oauth2-proxy como proxy para la segurización de otros servicios.

Y por último hemos probado colas de mensajes y como se podrían lanzar funciones a partir de otros eventos, para esto hemos probado los brokers nativos de Knative, eventos desde Minio a OpenFaaS, los conectores de OpenFaaS y Argo Events como event-driven.

Además de esto hemos desarrollado una pequeña prueba de concepto, que partiendo de las pruebas realizadas gestiona una pequeña aplicación de predicción meteorológica.

Introducción y objetivo

Desde que Amazon lanzó AWS Lambda se ha popularizado el uso de funciones como servicio (FaaS) por su facilidad de desarrollo y despliegue, por el ahorro de costes en mantenimiento de servidores, por su facilidad para escalar horizontalmente, y por el pago por ejecución de las funciones.

Tal es la implantación de esta solución que otros proveedores de cloud han implementado sus propios FaaS con características similares, como Azure Functions de Microsoft o Google Cloud Functions.

Pero para completar esta revolución en las aplicaciones de internet no solo se necesita poder ejecutar código también es necesario otros servicios que tengan ventajas similares.

Para esto los proveedores de cloud han ido proporcionando servicios como almacenamiento de ficheros como el famoso AWS S3 o su competencia en Azure y GCP.

También han proporcionado bases de datos autoescalables con pago por uso, como AWS DynamoDB, Azure Cosmos DB o Google Cloud Spanner, además de las bases de datos autoadministradas cómo AWS RDS.

Además de esto los proveedores de cloud, nos ofrecen sistemas para gestionar los usuarios de nuestra aplicación y segurizar los servicios que creemos, como AWS Cognito, Azure Active Directory B2C o Google Identity Platform.

Y para completar estos servicios nos ofrecen sistemas para ejecutar estas funciones por medio de colas de mensajes o otros eventos del cloud, como un push en un repositorio GIT, la creación o subida de un nuevo archivo en el almacenamiento de ficheros.

Pero todo esto tiene un inconveniente lo que se ha llamado el “vendor lock-in” qué básicamente es la imposibilidad o la gran dificultad de migrar de proveedor una vez tu aplicación esta desarrollada y funcionando. Si desarrollamos una aplicación con el stack serverless de AWS por ejemplo, haciendo uso de Lambda, S3, DynamoDB y Cognito, el coste de migrar el desarrollo “atado” a todos estos productos a otro proveedor de cloud en la mayoría de los casos será demasiado alto como para plantearlo.

Por esta razón y aprovechando que Kubernetes se ha convertido en un estándar en la industria de los servicios cloud, en este trabajo vamos a buscar y probar algunos servicios que puedan sustituir a los servicios nativos, en un entorno Kubernetes.

Partiendo del clúster de desarrollo que nos provee Microk8s vamos a probar operadores de bases de datos, almacenamiento de objetos, funciones como servicio, gestión de usuarios y como orquestar estos sistemas mediante colas y eventos.

Funciones como servicio (FaaS)

Como hemos comentado en el inicio las funciones como servicio es el origen de este ecosistema y es uno de los servicios que más alternativas tiene en Kubernetes.

Hay una gran cantidad de proyectos que implementan FaaS en Kubernetes como, por ejemplo, Knative[1], OpenFaaS[2], OpenWhisk[3], Kubeless[4], Fission[5], Fn[6].

Algunos son proyectos independientes como OpenFaaS, otros son proyectos mantenidos por grandes empresas como Oracle FN, casi todos, por no decir todos soportan gran cantidad de lenguajes de programación para desarrollar las funciones y la mayoría están basados en el uso de contenedores para desplegar las funciones en Kubernetes.

Hemos realizado la instalación y pruebas sobre 2 de ellas OpenFaaS y Knative, aunque las pruebas de integración con el resto de los servicios del ecosistema se han realizado sobre OpenFaaS.

[1] https://github.com/knative/serving
[2] https://github.com/openfaas/faas
[3] https://github.com/apache/openwhisk
[4] https://github.com/kubeless/kubeless
[5] https://github.com/fission/fission
[6] https://github.com/fnproject/fn

Knative

Knative es un proyecto de FaaS desarrollado por google y liberado como opensource. El sistema de Knative se consta de 2 componentes "Knative Serving" y "Knative Eventing", el primero son las funciones propiamente dichas, y el segundo es el sistema para ejecutarlas en función de varios eventos.

Todos los componentes de Knative se despliegan en el clúster Kubernetes como "custom resource definitions" o CRDs. Estos CRDs se pueden desplegar, con un fichero yaml con el cliente kubectl o con el cliente propio de Knative kn.

Instalación

La instalación de Knative se realiza desplegando sus componentes mediante manifiestos Kubernetes.

```
kubectl apply --filename https://github.com/knative/serving/releases/downloa
d/v0.15.0/serving-crds.yaml
kubectl apply --filename https://github.com/knative/serving/releases/downloa
d/v0.15.0/serving-core.yaml
kubectl apply --filename https://github.com/knative/net-istio/releases/downl
oad/v0.15.0/release.yaml
kubectl apply --filename https://github.com/knative/eventing/releases/downlo
ad/v0.16.0/eventing-crds.yaml
kubectl apply --filename https://github.com/knative/eventing/releases/downlo
ad/v0.16.0/eventing-core.yaml
kubectl apply --filename https://github.com/knative/eventing/releases/downlo
ad/v0.16.0/in-memory-channel.yaml
kubectl apply --filename https://github.com/knative/eventing/releases/downlo
ad/v0.16.0/mt-channel-broker.yaml
kubectl apply --filename https://github.com/knative/serving/releases/downloa
d/v0.16.0/monitoring-core.yaml
kubectl apply --filename https://github.com/knative/serving/releases/downloa
d/v0.16.0/monitoring-metrics-prometheus.yaml
```

Desarrollo y despliegue

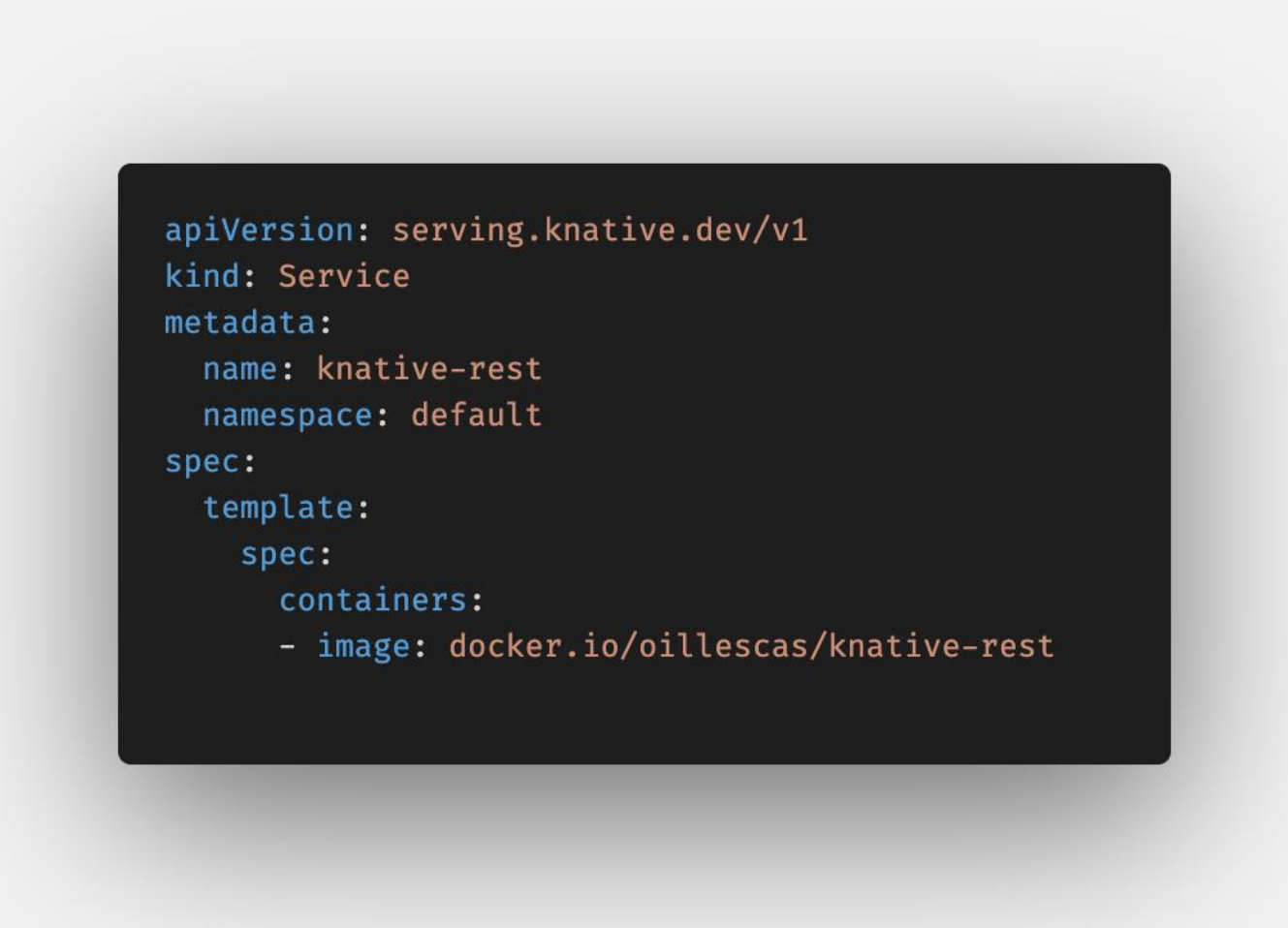
Las funciones se desarrollan y se empaquetan como cualquier otro contenedor Docker y se despliegan como un pod en el clúster Kubernetes a partir de un CRD.

Ejemplos

- Api Rest con Express simulando la base de datos en memoria aquí podemos ver el Código[7]

[7] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/tree/main/Examples/knative/REST

Y este es el manifiesto para desplegar ese contenedor como una función de Knative.



```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: knative-rest
  namespace: default
spec:
  template:
    spec:
      containers:
      - image: docker.io/oillescas/knative-rest
```

OpenFaaS

OpenFaaS es el servicio de FaaS que más estrellas tiene en github a fecha de 20 de noviembre de 2020 tiene unas 18.800 está escrito en su mayor parte en go.

Instalación

La instalación de OpenFaaS se realiza clonando el repositorio de OpenFaaS, creando un secreto y de Kubernetes para el usuario y la password de gestión de OpenFaaS, y para terminar desplegando los manifiestos de Kubernetes.

```
git clone https://github.com/openfaas/faas-netes

# generate a random password
PASSWORD=$(head -c 12 /dev/urandom | shasum| cut -d' ' -f1)

kubectl -n openfaas create secret generic basic-auth \
--from-literal=basic-auth-user=admin \
--from-literal=basic-auth-password="$PASSWORD"

cd faas-netes && \
kubectl apply -f namespaces.yml
kubectl apply -f ./yaml
```

Plantillas

OpenFaaS funciona a partir de plantillas para generar los contenedores, gestiona un repositorio oficial con platillas8 para muchos lenguajes de programación. Además de esto también es fácil la creación de nuevas plantillas para añadir nuevos lenguajes o para abstraer al desarrollador de tareas comunes, librerías de terceros, ...

Hemos creado un repositorio[9] para implementar algunos ejemplos de plantillas.

- Nodejs12-files
- Nodejs12-nats

Estas plantillas están basadas en la plantilla original Nodejs12[10] que usa un servidor Express para gestionar las llamadas http.

[8] https://github.com/openfaas/templates
[9] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes-openfass
[10] https://github.com/openfaas/templates/tree/master/template/node12

Nodejs12-files

En esta template hemos añadido el plugin de express file-upload[11] para permitir que se puedan subir archivos vía http.

```
const fileUpload = require('express-fileupload');
// enable files upload
app.use(fileUpload({
    createParentPath: true,
    limits: {
        fileSize: 100 * 1024 * 1024 * 1024
//100MB max file(s) size
    },
    debug: true
}));
```

[12]

Además, en el fichero dockerfile[13] hemos aumentado los tiempos máximos de ejecución.

```
# ENV exec_timeout="10s"
# ENV write_timeout="15s"
# ENV read_timeout="15s"

ENV exec_timeout="60s"
ENV write_timeout="40s"
ENV read_timeout="40s"
```

[11] https://www.npmjs.com/package/express-fileupload
[12] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes-openfass/blob/main/template/node12-files/index.js
[13] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes-openfass/blob/main/template/node12-files/Dockerfile#L56

Nodejs12-nats

En esta template, como ejemplo, hemos modificado la gestión que hace la plantilla original del body http, para recibir los mensajes de una cola nats. [14]

```
// if (process.env.RAW_BODY === 'true') {
//     app.use(bodyParser.raw({ type: '*/*' }))
// } else {
//     var jsonLimit = process.env.MAX_JSON_SIZE || '100kb' //body-parser default
//     app.use(bodyParser.json({ limit: jsonLimit}));
//     app.use(bodyParser.raw()); // "Content-Type: application/octet-stream"
//     app.use(bodyParser.text({ type : "text/*" }));
// }

app.use (function(req, res, next) {
    var data='';
    req.setEncoding('utf8');
    req.on('data', function(chunk) {
        data += chunk;
    });

    req.on('end', function() {
        req.body = data;
        next();
    });
});
```

Desarrollo y despliegue

Para desarrollar una función OpenFaaS hay que instalar el template desde el cual se quiere partir utilizando el cli. Podemos descargar las plantillas oficiales con el siguiente comando:

faas template pull

O podemos descargar plantillas de cualquier otro repositorio con el comando:

faas template pull https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes-openfass

Una vez descargadas las plantillas creamos una nueva función con el cli.

faas new --lang java11 hello-java

[14] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes-openfass/blob/main/template/node12-nats/index.js#L11

Esto nos genera una carpeta con el “handler” en este caso un fichero js que expone una función, o en java seria una clase que extiende de “com.openfaas.model.AbstractHandler”, este es el fichero donde vamos a codificar la lógica de nuestra función y el archivo correspondiente a la gestión de dependencias de cada lenguaje en caso de java “*build.gradle”* o en caso de JavaScript “packaje.json”.

Además, encontraremos el descriptor de despliegue de la función que acabamos de crear “*stack.yml*”.

```
version: 1.0
provider:
  name: openfaas
functions:
  hello-java:                          //Nombre de la función
    lang: java11                       //Planilla de la que partimos
    handler: ./hello-java              //Path del handler
    image: oillescas/hello-java:latest  //Nombre de la imagen docker que se creará
```

Ejemplos

Hola mundo Java

En este caso la función escrita en Java siempre devuelve un “Hello, world!”.

```java
package com.openfaas.function;

import com.openfaas.model.IHandler;
import com.openfaas.model.IResponse;
import com.openfaas.model.IRequest;
import com.openfaas.model.Response;

public class Handler extends com.openfaas.model.AbstractHandler
 {

    public IResponse Handle(IRequest req) {
        Response res = new Response();
        res.setBody("Hello, world!");

        return res;
    }
}
```

Hola mundo JavaScript

En este ejemplo vemos el código js de una función simple que siempre devuelve un código 200 y un json que devuelve el body original de la petición.

```javascript
'use strict'

module.exports = async (event, context) => {
  const result = {
    'status': 'Received input: ' + JSON.stringify(event.body)
  }

  return context
    .status(200)
    .succeed(result)
}
```

Gestión de ficheros

En este capítulo intentamos encontrar una alternativa al almacenamiento de objetos como AWS S3, Azure Blob Storage o Google Cloud Storage.

Minio

Hemos elegido Minio que implementa una interfaz compatible con AWS S3 y que es escalable gracias a su operador[15]. Por simplicidad y por que estamos en un entorno de desarrollo como Microk8s hemos usado el chart de helm para desplegar una sola instancia de Minio.

Como ventaja añadida Minio permite usarse como proxy de los servicios nativos de AWS y Azure, entre otros, lo que permite usarlo dentro de un clúster Kubernetes aprovechando las capacidades de los servicios nativos de los proveedores de cloud.

Instalación

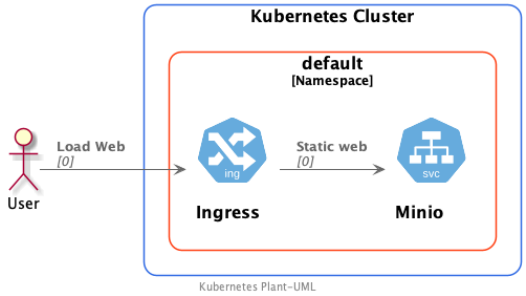
En este caso la instalación de Minio la hemos realizado con helm con las siguientes instrucciones:

```
#Instalación con helm
helm repo add minio https://helm.min.io/
helm install --set service.type=LoadBalancer my-minio minio/minio
```

[15] https://github.com/minio/operator

Ejemplos

Despliegue de un sitio estático



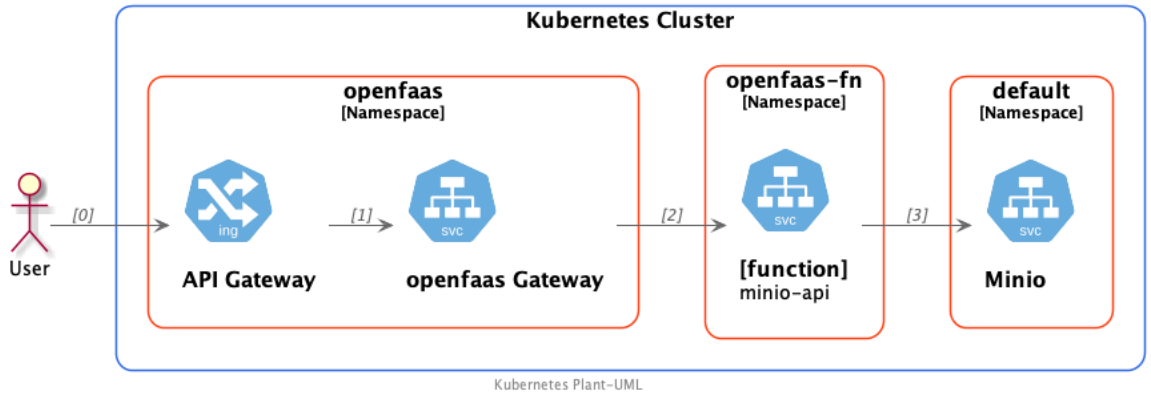
Una de las funcionalidades que se le dan a este tipo de servicios es almacenar sitios web. Para esto necesitamos crear un bucket en minio y subir los archivos al mismo. Además de esto es necesario configurar la policy de download para este bucket. Aquí[16] se puede encontrar las instrucciones detalladas para esto.

Una vez configurado el bucket de Minio necesitamos configurar un Ingress NGINX de Kubernetes para servir los ficheros.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simpleapp-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: "/static-site/$1"
    nginx.ingress.kubernetes.io/configuration-snippet: |
      rewrite ^/?$ /static-site/index.html break;
spec:
  # ingressClassName: nginx
  rules:
  - host: miniostatic.192.168.0.100.nip.io
    http:
      paths:
      - path: /(.*)
        pathType: Prefix
        backend:
          service:
            name: my-minio
            port:
              number: 9000
```

[16] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/tree/main/2.GestionArchivos/minio#despliegue-de-un-sitio-est%C3%A1tico

Api minio con OpenFaaS



En este ejemplo implementamos un pequeño api para interactuar con minio mediante una función OpenFaaS escrita en Node.js con una de las custom templates que hemos presentado en la parte de OpenFaaS.

Aquí[17] podemos ver el código de la función, en este caso Minio tiene su propia librería y hacemos uso de esta para acceder a los datos. También usamos secretos de Kubernetes para guardar las claves de acceso a Minio. Y aquí podemos ver el descriptor de la función donde vemos las variables de entorno para configurar el servidor de minio y como se usan los secretos de Kubernetes en los descriptores OpenFaaS.

```yaml
version: 1.0
provider:
  name: openfaas
functions:
  openfaas-minio-api:
    lang: node12-files
    handler: ./minio-api
    image: oillescas/minio-api:latest
    environment:
      endpoint: my-minio.default.svc.cluster.local
      minio-port: 9000
      use-ssl: false
    secrets:
    - openfaas-minio
```

[17] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/blob/main/Examples/openfaas/minio-api/handler.js

Bases de datos

En este punto vamos a buscar alternativas a las bases de datos no relacionales y autoescalables de las plataformas en la nube como AWS Dynamodb, Azure Cosmos DB o Google Cloud Spanner.

Buscando entre varios operadores de motores de base de datos como MongoDB, mysql o PostgreSQL se han elegido 2 operadores por su simplicidad de instalación. Un operador de base de datos no relacional como MongoDB y un operador de bases de datos relacionales como PostgreSQL.

MongoDB

En este caso hemos probado el operador de MongoDB de Percona LLC[18].

Instalación

La instalación de esto operador se limita a aplicar un manifiesto Kubernetes

```
# Despliegue del operador
kubectl apply -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/v1.7.0/deploy/bundle.yaml
```

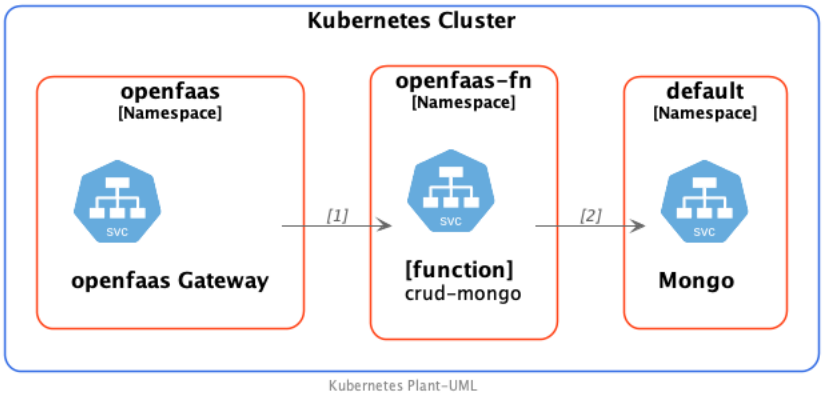
Despliegue de un clúster MongoDB

Para desplegar un clúster MongoDB usando el operador de Percona debemos desplegar en Kubernetes un nuevo CRD de tipo *PerconaServerMongoDB* al contrario que la instalación que es muy simple el manifiesto de este CRD es bastante complejo y podemos ver un ejemplo en el repositorio[19] de este trabajo junto con las instrucciones[20] para probar su funcionamiento.

[18] https://www.percona.com/
[19] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/blob/main/3.BasesDeDatos/perconaMongodb/custom_cr.yaml
[20] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/tree/main/3.BasesDeDatos/perconaMongodb#creamos-nuestro-cluster-mongodb

Ejemplos

Crud con MongoDB y OpenFaaS



En este ejemplo hemos implementado un api rest para gestionar usuarios y comentarios conectando al clúster MongoDB creado anteriormente.

Podemos ver el código y el descriptor de la función en el repositorio[21].

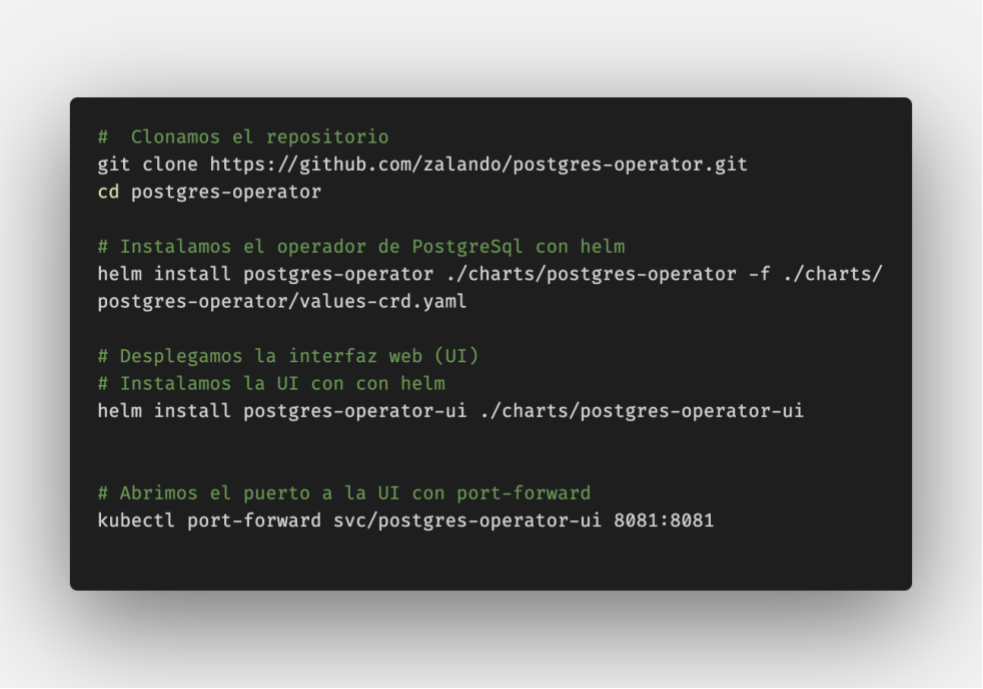
[21] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/tree/main/Examples/openfaas/crud-mongo

PostgreSQL

También hemos probado el operador de PostgreSQL de Zalando.

Instalación

En el caso del operador de PostgreSQL de Zalando, la instalación se realiza mediante un chart de helm y además del operador opcionalmente podemos instalar una interfaz grafica para administrar los clúster PostgreSQL



```
#  Clonamos el repositorio
git clone https://github.com/zalando/postgres-operator.git
cd postgres-operator

# Instalamos el operador de PostgreSql con helm
helm install postgres-operator ./charts/postgres-operator -f ./charts/
postgres-operator/values-crd.yaml

# Desplegamos la interfaz web (UI)
# Instalamos la UI con con helm
helm install postgres-operator-ui ./charts/postgres-operator-ui


# Abrimos el puerto a la UI con port-forward
kubectl port-forward svc/postgres-operator-ui 8081:8081
```

Despliegue de un clúster PostgreSQL

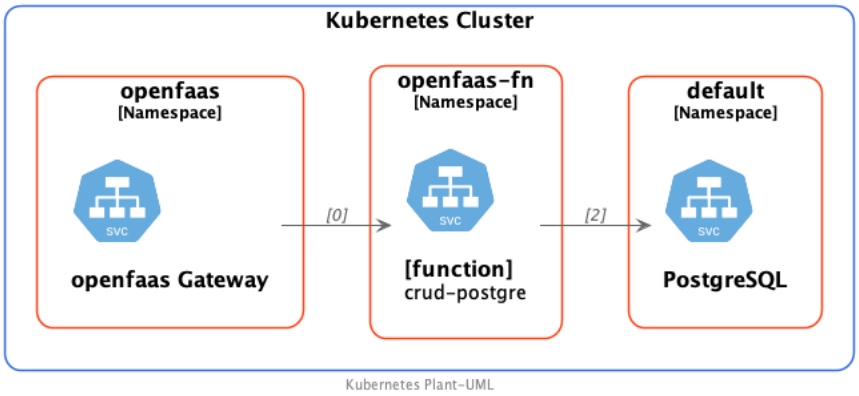
El caso del operador de PostgreSQL debemos desplegar un CRD de tipo PostgreSQL con un manifiesto bastante mas sencillo que el de MongoDB. Igualmente podemos encontrar el manifiesto[22] y las instrucciones[23] para probarlo en el repositorio de este trabajo.

[22] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/blob/main/3.BasesDeDatos/zalandoPostgresOperator/minimal-postgres-manifest.yaml
[23] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/blob/main/3.BasesDeDatos/zalandoPostgresOperator/readme.md#creamos-nuestro-cluster-postgresql

```yaml
apiVersion: "acid.zalan.do/v1"
kind: postgresql
metadata:
  name: acid-minimal-cluster
  namespace: default
spec:
  teamId: "acid"
  volume:
    size: 1Gi
  numberOfInstances: 2
  users:
    zalando:  # database owner
    - superuser
    - createdb
    foo_user: []  # role for application foo
  databases:
    foo: zalando  # dbname: owner
  preparedDatabases:
    bar: {}
  postgresql:
    version: "12"
```

Ejemplos

REST api PostgreSQL con OpenFaaS



En este caso probamos con una función OpenFaaS que implementa una gestión simple de dispositivos IOT la conexión al clúster de PostgreSQL que hemos creado anteriormente.

El código y el descriptor se pueden ver en el repositorio[24].

[24] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/tree/main/Examples/openfaas/crud-postgre

Colas y gestión de eventos

Otra parte importante del ecosistema serverless es como y cuando se lanzan las funciones, hasta ahora todas las funciones se han invocado vía http.

Channels Knative

El sistema Knative implementa un sistema de eventos mediante "canales" para publicar y consumir eventos o mediante "event broker" para recibir y distribuir los eventos. La capa de canales tiene varias implementaciones sobre colas de mensajes como Kafka, Nats, Google cloud pub/sub y la que hemos probado una implementación en memoria. Además, soporta extensiones para gestionar eventos desde otros sistemas como github o Apache Kamel entre otros.

Ejemplo

En este ejemplo hemos echo uso del event broker en memoria de Knative, lanzar un evento al broker y comprobar como se ejecutan uno de los dos servicios que hemos desplegado. Desplegamos un Broker y dos Trigger que lanzan una petición a dos servicios.

En el repositorio[25] podemos ver las instrucciones para instalar y probar este ejemplo.

```yaml
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: default
  namespace: event-example

---

apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: hello-display
spec:
  broker: default
  filter:
    attributes:
      type: greeting
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: hello-display

---

apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: goodbye-display
spec:
  broker: default
  filter:
    attributes:
      source: sendoff
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: goodbye-display
```

[25] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/blob/main/4.ColasEventos/Knative/events.md

Conectores en OpenFaaS

En OpenFaaS las funciones además de vía http pueden ejecutarse desde otros eventos, pero necesitan una pequeña pieza de código que denominan "connector". Dentro del proyecto OpenFaaS se soportan varios conectores como nats-connector para colas nats, o el cron-connector para programación de tareas, pero además hay conectores desarrollados por terceros para colas MQTT, colas Kafka, AWS SNS, AWS SQS redis, incluso IFTTT. En este trabajo hemos probado el nats-connector y el cron-connector, además de ejecutar una función como webhook de Minio.

nats-connector

En el proceso de instalación de OpenFaaS además de el propio sistema de funciones se instala una cola de mensajes Nats y contra esta cola hemos probado 2 funciones que publican y reciben mensajes, mediante el conector nats-connector[26].

Ejemplo

En este ejemplo desplegamos el "nats-connector" conectado a la cola nats que instala el propio OpenFaaS, desplegamos una función que añade un mensaje a la cola, y otra función que se ejecuta al recibir un mensaje de esta cola.

En el descriptor de la función que se ejecuta con el mensaje definimos el "topic" que debe escuchar.

```
version: 1.0
provider:
  name: openfaas
functions:
  nats-reciver:
    lang: node12-nats
    handler: ./nats-reciver
    image: oillescas/nats-reciver:latest
    annotations:
      topic: "nats-test"
```

[26] https://github.com/openfaas/nats-connector

cron-connector

OpenFaaS nos provee de otros conectores como el cron-connector[27] que nos provee de una manera de ejecutar funciones de una manera programada como podemos ver en el ejemplo download-forecas

Ejemplo

En este ejemplo desplegamos el "cron-connector" y desplegamos una función OpenFaaS con las anotaciones necesarias para que se ejecute automáticamente cada 4 horas.

```yaml
version: 1.0
provider:
  name: openfaas
functions:
  download-forecast:
    lang: node12
    handler: ./download-forecast
    image: oillescas/download-forecast:latest
    annotations:
      topic: cron-function
      schedule: "1 */4 * * *"
    environment:
      endpoint: my-minio.default.svc.cluster.local
      minio-port: 9000
      use-ssl: false
      BUCKET_NAME: forecast
    secrets:
    - openfaas-minio
```

[27] https://github.com/openfaas/cron-connector

Minio

Minio como sistema de gestión de gestión de ficheros puede enviar eventos cuando se producen cambios en los ficheros, y para probar esto podemos ver el ejemplo "minio-webhook[28]" que lanza una ejecución de una función OpenFaaS cuando se añade un nuevo archivo.

Ejemplo

En este caso hay que añadir la configuración necesaria con "minio client" para que llame a una función OpenFaaS
Primero configuramos un endpoint de tipo webhook

mc admin config set minio-tfm notify_webhook:1
endpoint=http://gateway.openfaas:8080/function/minio-webhook

y a continuación definimos con que evento queremos que se lance ese webhook

mc event add minio-tfm/forecast arn:minio:sqs::1:webhook --event put --suffix .xml

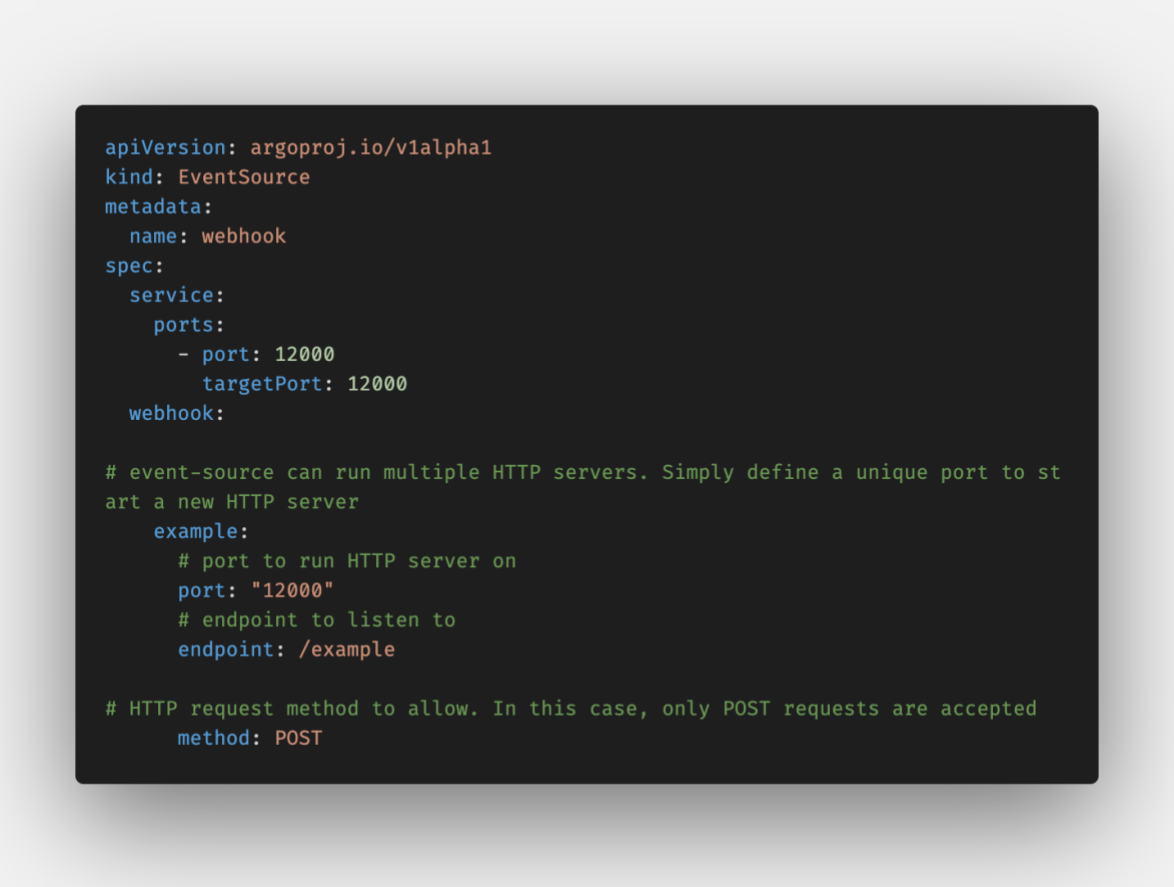
[28] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/blob/main/4.ColasEventos/OpenFaaS/events.md#minio-webhook

Argo Events

Además de las funcionalidades que añaden los propios sistemas de FaaS, podemos usar un sistema de un tercero para orquestar las ejecuciones de las funciones.
En este caso vamos a instalar y probar Argo Events[29], que soporta mas de 20 orígenes de eventos[30] entre los que se encuentran por ejemplo minio, s3, cron, github, gitlab, slack, webhooks, AMPQ.

Ejemplo

Como prueba hemos desplegado un eventSource de tipo webhook que recibirá las peticiones



```yaml
apiVersion: argoproj.io/v1alpha1
kind: EventSource
metadata:
  name: webhook
spec:
  service:
    ports:
      - port: 12000
        targetPort: 12000
  webhook:

# event-source can run multiple HTTP servers. Simply define a unique port to st
art a new HTTP server
    example:
      # port to run HTTP server on
      port: "12000"
      # endpoint to listen to
      endpoint: /example

# HTTP request method to allow. In this case, only POST requests are accepted
      method: POST
```

[29] https://argoproj.github.io/argo-events/
[30] https://argoproj.github.io/argo-events/concepts/event_source/

y un sensor que mediante un trigger las conducirá vía http a una función OpenFaaS.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Sensor
metadata:
  name: openfaas-sensor
spec:
  template:
    serviceAccountName: argo-events-sa
  dependencies:
    - name: test-dep
      eventSourceName: webhook
      eventName: example
  triggers:
    - template:
        name: openfaas-trigger
        http:
          url: http://gateway.openfaas.svc.cluster.local:8080/function/hello-world
          payload:
            - src:
                dependencyName: test-dep
              dest: bucket
          method: POST
```

Gestión de usuarios

También necesitamos gestionar usuarios y para esto hemos buscado alternativas a AWS Cognito, Azure Active Directory o Google Identity Platform.

Keycloak

En este caso hemos probado Keycloak que es una solución de gestión de acceso e identidad que soporta entre SAML v2 y oauth2. Y lo hemos combinado con el proyecto Oauth2-proxy[31] para segurizar algunos servicios

Instalación

La instalación básica de Keycloak se realiza mediante un solo manifiesto de Kubernetes.

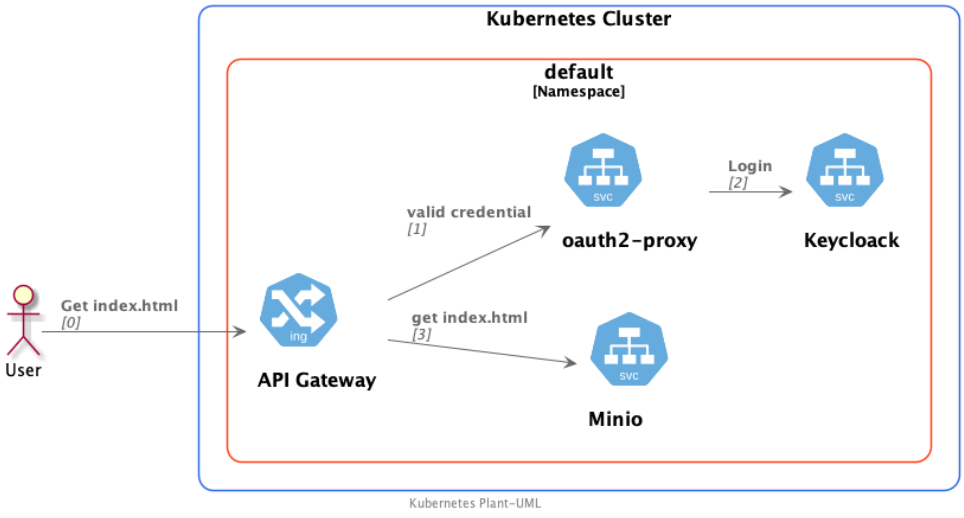
```
# Instalamos Keycloak.

kubectl create -f https://raw.githubusercontent.com/keycloak/keycloak-quickstarts/latest/kubernetes-examples/keycloak.yaml
```

Una vez instalado podemos seguir las instrucciones[32] que se describen en el repositorio para importar la configuración necesaria para probar los ejemplos.

Ejemplos

Ingress con el que servimos las páginas estáticas con Minio.



[31] https://oauth2-proxy.github.io/oauth2-proxy/

[32]

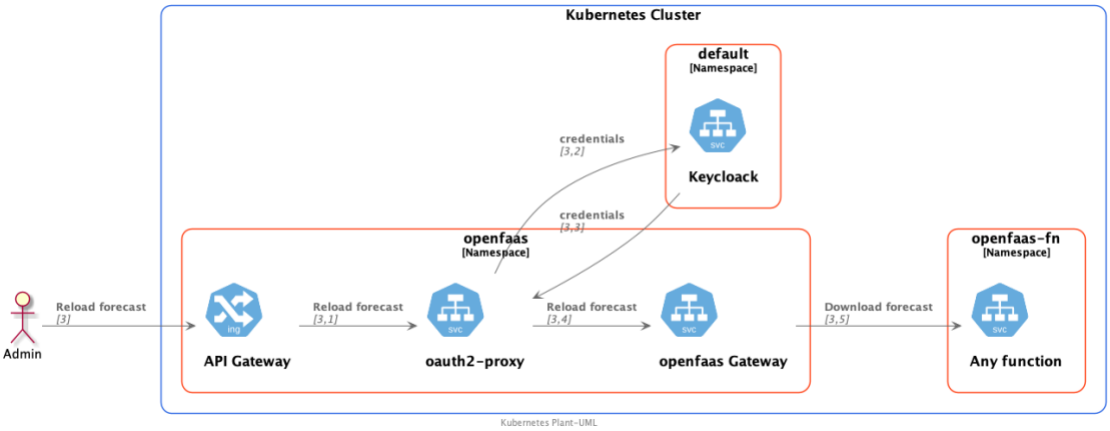
Con un Ingress NGINX y el auth2-proxy podemos segurizar las peticiones que se realicen a ese Ingress mediante las siguientes anotaciones:
nginx.ingress.kubernetes.io/auth-url
nginx.ingress.kubernetes.io/auth-signin

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simpleapp-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: "/static-site/$1"
    nginx.ingress.kubernetes.io/configuration-snippet: |
      rewrite ^/?$ /static-site/index.html break;
    nginx.ingress.kubernetes.io/auth-url:
"https://$host/oauth2/auth"
    nginx.ingress.kubernetes.io/auth-signin:
"https://$host/oauth2/start?rd=$escaped_request_uri"
spec:
# ....
```

Y configurando los endpoints para que usen el Oauth2-proxy, En el repositorio[33] se pueden encontrar todas las configuraciones necesarias.

[33] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/tree/main/5.Usuarios/Keycloak/simple-ingress

Función de OpenFaaS



De la misma manera que el ejemplo anterior podemos segurizar la llamada a una función OpenFaaS, aunque en este caso lo que hacemos es exponer el servicio del Oauth2-proxy vía Ingress y hacer que este realice las llamadas a el Gateway de OpenFaaS.

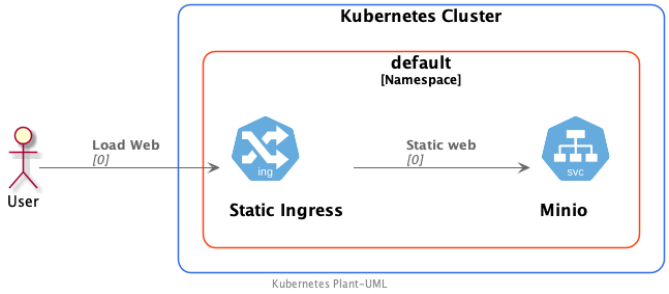
Como en todos los ejemplos se puede encontrar el código en el repositorio[34], y podemos ver como segurizamos todas las funciones OpenFaaS instaladas.

[34] https://github.com/MasterCloudApps-Projects/Serverless-Kubernetes/tree/main/5.Usuarios/Keycloak/OpenFaaS

Aplicación

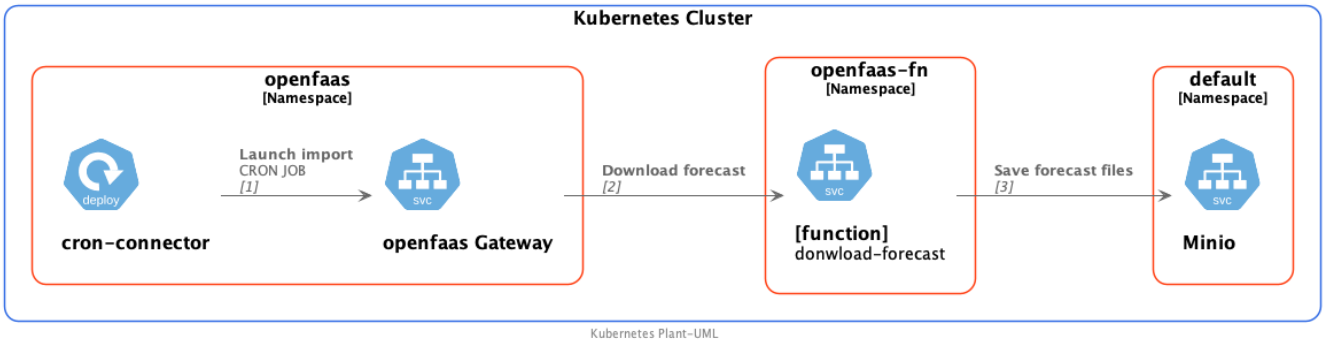
Para probar como se integran algunas de las funcionalidades que hemos analizado hemos desarrollado una aplicación completa.
Que muestra una pequeña predicción meteorológica extraída del api de AEMET, de un numero reducido de localidades.

Que tiene un pequeño frontal html que se sirve mediante Minio y un Ingress.



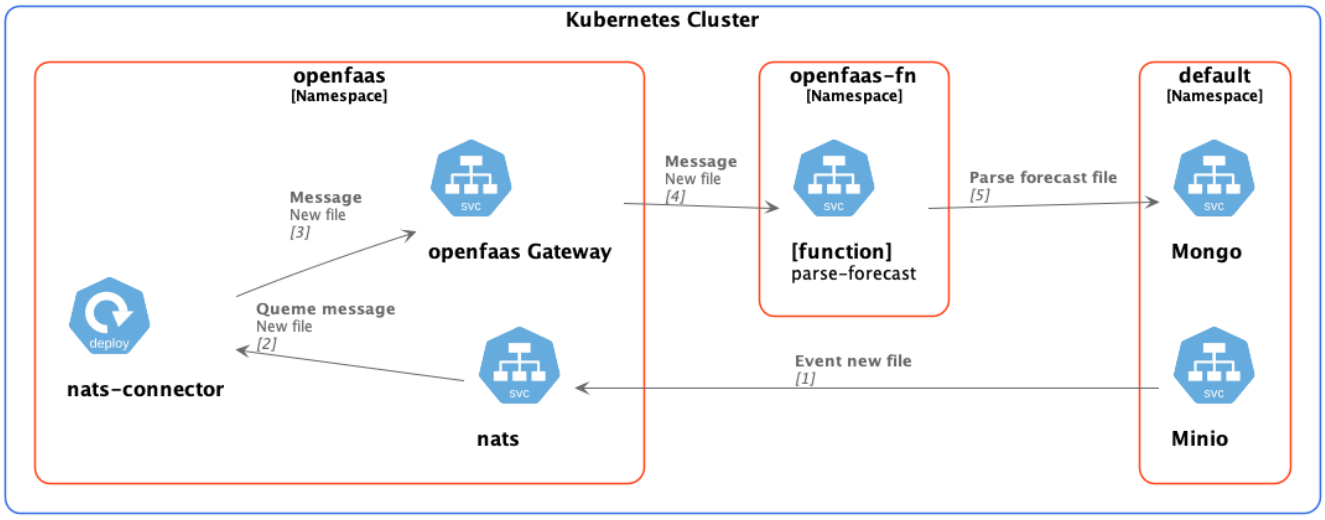
A su vez tenemos varias funciones implementadas con OpenFaaS que se encargan de extraer transformar y servir las predicciones.

Función donwload-forecast



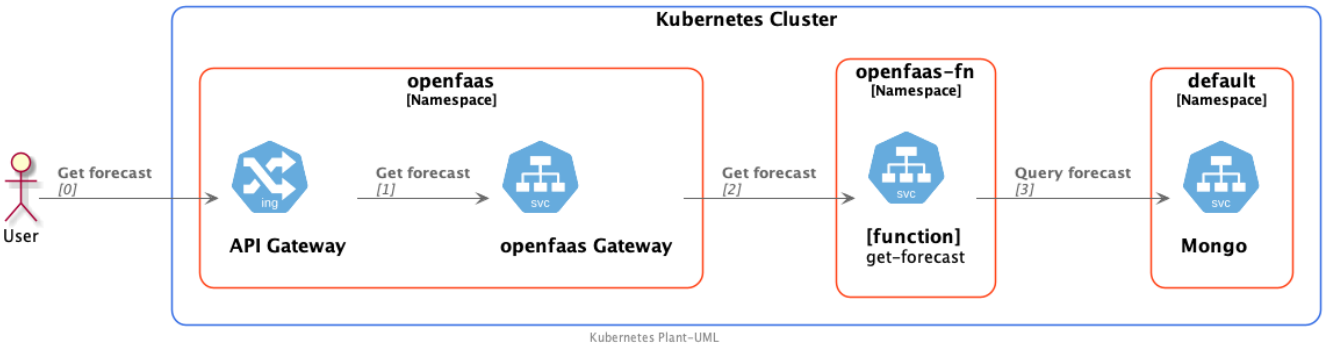
Esta función hace uso del cron-connector para consultar el api de AEMET y descargar las previsiones meteorológicas y guardarlas en minio, para su posterior consulta.

Función parse-forecas



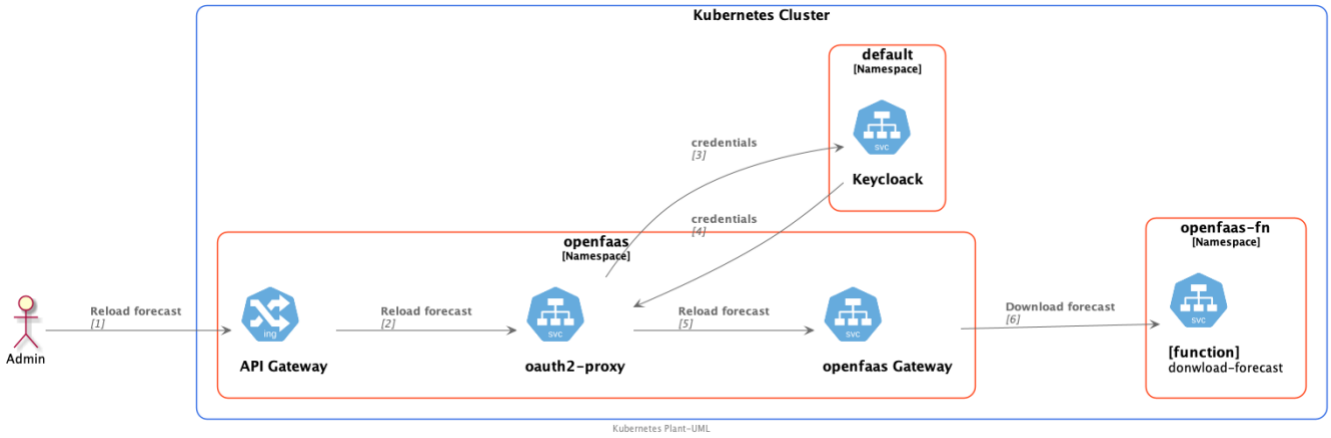
Esta funcionalidad hace uso de los eventos de minio para lanzar un evento put hacia una cola nats que gracias al nats-connector lanza la función parse-forecas.
Esta función consulta la predicción guardada en minio, genera un objeto con los datos útiles de la predicción y los guarda en la base de datos MongoDB.

Función get-forecast

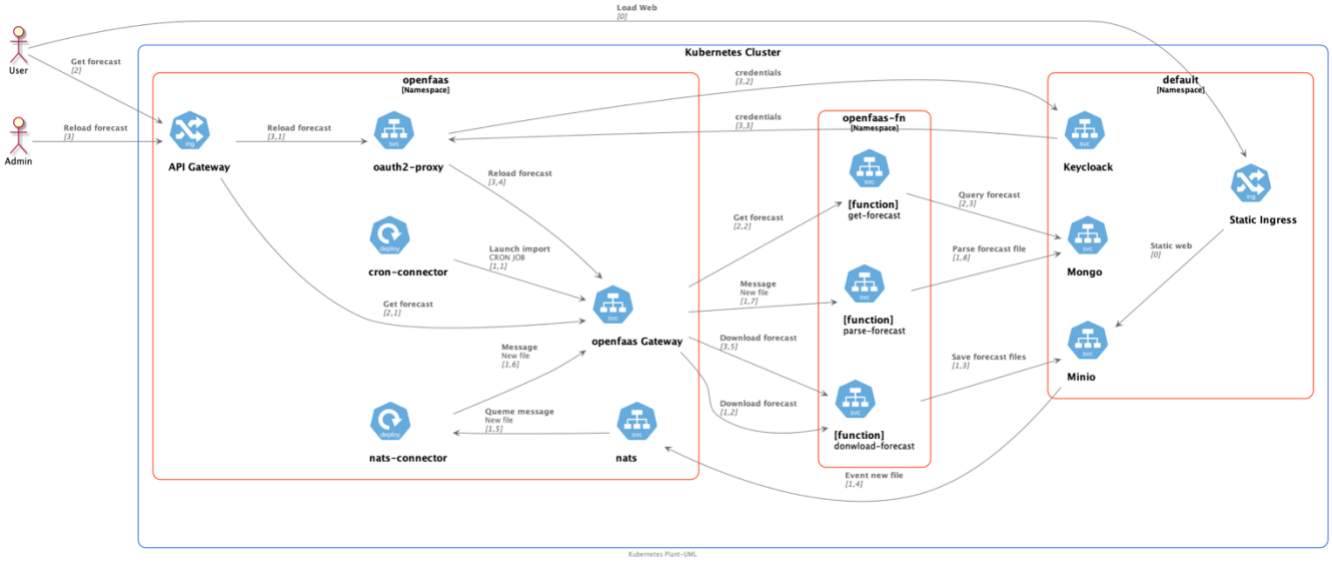


Esta ultima función se encarga de servir las consultas del cliente HTML con la información guardada en la base de datos MongoDB

Además, hemos creado un punto de acceso segurizado para que un usuario administrador pueda forzar la recarga de las predicciones.



Para finalizar aquí podemos ver el diagrama completo de la aplicación.



Conclusiones y Trabajos futuros

Conclusiones

Después de instalar, estudiar y probar estos servicios podemos llegar a la conclusión de que es posible sustituir el stack serverless que nos sirven los proveedores de cloud por un stack desplegado en un clúster Kubernetes.

Hemos comprobado que existen alternativas escalables para el almacenamiento de ficheros, como Minio. También hemos visto que hay varios operadores de bases de datos que nos dan un servicio equivalente al de los proveedores de cloud, además con la ventaja de que además de bases de datos noSQL podemos tener bases de datos relacionales.

Por otro lado, vemos que hay varias opciones para las FaaS, además de las probadas, que nos permiten escalabilidad y monitorización y que podemos ejecutar estas funciones vía colas y eventos tanto con las integraciones nativas de las implementaciones de FaaS como con herramientas externas del ecosistema Kubernetes.

Además de lo anterior en general todos los servicios que hemos probado son sustituibles por sus equivalentes nativos de los proveedores de cloud, por ejemplo, podría montar una aplicación que se ejecute en un clúster Kubernetes ejecutando OpenFaaS y usar AWS S3 y AWS DynamoDB para el almacenamiento de ficheros y datos. Esto nos "ataría" a AWS pero no de la misma manera que si uso todo el stack de Amazon.

Conclusiones personales.

Personalmente este trabajo me ha permitido adentrarme en el uso de los clúster de Kubernetes, que prácticamente desconocía antes de comenzar el máster y también analizar las posibilidades del stack serverless tanto nativo como en Kubernetes con el que, si había trabajado, pero nunca había visto el potencial que puede llegar a tener.

Al analizar los resultados me surge un inconveniente a la idea de implementar serverless en Kubernetes, ya que una de las ventajas del stack Serverless de los proveedores de cloud es el coste, estos servicios se facturan por uso, y si no hay uso no hay gasto. Mientras que para tener un clúster Kubernetes necesitas una mínima infraestructura siempre funcionado e ir aumentándola al ritmo que crece el uso de los servicios.

Por esto montar este serverless sobre Kubernetes me parece buena opción para una organización que ya disponga de infraestructura propia y quiera migrar o empezar con el stack serverless, pero no tanto para una organización que empieza de cero.

Trabajos futuros

Como he comentado anteriormente tengo mis dudas sobre el coste de esta solución sobre Kubernetes en comparación con los costes de los servicios propietarios de los proveedores de cloud, y por esta razón me gustaría comparar y estudiar los costes entre las soluciones propietarias y la solución Kubernetes

Además, durante tiempo que he estado buscando alternativas a los servicios propietarios me he encontrado con muchos mas proyectos que me gustaría conocer.
Como otros operadores de base de datos:
El operador oficial de MongoDB, Enterprise MongoDB[35] o Scylla Alternator[36] que implementa una interfaz compatible con AWS DynamoDB.

También he encontrado herramientas para orquestar ejecuciones de funciones serverles encadenarlas y paralelizarlas, tanto ajenas a los frameworks de FaaS, como puede ser Argo Workflows[37] o diseñadas y desarrolladas para los propios frameworks como Faas Flows[38] para OpenFaaS

Y para terminar me gustaría adentrarme más en el ecosistema de OpenFaaS con otros desarrollos enfocados a facilitar el uso de este sistema de FaaS como "OpenFaaS Ingress Operator[39]" un operador para ingress que facilita la forma de exponer las funciones fuera del contenedor Kubernetes o OpenFaaS cloud[40] que es un sistema de gestión de funciones que entre otras cosas permite CI/CD e integración con GitLab y GitHub.

[35] https://github.com/mongodb/mongodb-enterprise-kubernetes
[36] https://docs.scylladb.com/using-scylla/alternator/
[37] https://argoproj.github.io/projects/argo
[38] https://github.com/s8sg/faas-flow
[39] https://github.com/openfaas/ingress-operator
[40] https://github.com/openfaas/openfaas-cloud

Bibliografía

- https://en.wikipedia.org/wiki/Serverless_computing#Serverless_runtimes
- https://www.percona.com/doc/kubernetes-operator-for-psmongodb/index.html
- https://www.fundeu.es/recomendacion/segurizar-securizar-securitizar/
- https://vshn.ch/en/blog/a-very-quick-comparison-of-kubernetes-serverless-frameworks/