Despliegue de un sistema con CrateDB

Joel Bra Ortiz

2020–12–10

1 Introducción

En este proyecto se nos pide aplicar los conocimientos recibidos en la asignatura de Administración de Sistemas para gestionar y orquestar un conjunto de contenedores.

En primer lugar, se nos ha provisto de una imagen docker de Docker Hub que tendremos que utilizar como base, en este caso, CrateDB.

Por otro lado, deberemos incluir otras funcionalidades a la aplicación en forma de contenedor y generar un despliegue conjunto con docker-compose.

2 Los contenedores

2.1 CrateDB

CrateDB es el contenedor que me ha sido asignado. Es una base de datos SQL construida sobre una arquitectura en la nube. Además, añade la escalabilidad y flexibilidad de una base de datos NoSQL.

Está escrita en Java y basada en la arquitectura shared-nothing, la cual consiste en que cada nodo es autosuficiente e independiente. De esta forma, ningún nodo comparte memoria o espacio de disco duro.

CrateDB cuenta con unos nodos que distribuyen y coordinan de manera automática la ejecución de las tareas de escritura y peticiones a lo largo del cluster.

Al tratarse de una base de datos SQL tiene a nuestra disposición JOINS, índices, vistas…Para su distribución, CrateDB cuenta con una cahé que le dicen al motor de peticiones si existe alguna fila que cumpla con los criterios de la query.

Por otro lado, CrateDB soporta esquemas de tipo estricto, dinámico o ignorado. Al igual que una base de datos noSQL, permite actualizar o no el esquema de una tabla al realizar un INSERT.

CrateDB implementa un sistema que se contrapone al clásico sistema ACID (atomicity, consistency, isolation, durability), siendo clasificado como BASE (basically-available, soft-state, eventual consistency). Cuenta con un sistema de versiones, un sistema de concurrencia optimista (asume que varias transacciones se pueden realizar sin interferir entre ellas) y un ajuste de actualización a nivel de tabla que le permite ser consistente cada x milisegundos.

En cuanto a la atomicidad y la durabilidad, las operaciones sobre filas son atómicas. Además, éstas tendrán persistencia en disco sin realizar un commit. En caso de que se produzca un apagado inesperado de un nodo, las operaciones del diario se vuelven a ejecutar para asegurarse de que todas las operaciones se han realizado.

2.2 Aplicación cliente

La aplicación conocida como cliente hará las veces de aplicación servidor que nos conectará con la base de datos CrateDB para almacenar la información pertinente y que esta se visualice en el frotnend.

Para la programación de la misma se ha utilizado el lenguaje Python que nos permite, por un lado, recibir peticiones mediante la librería *Flask* y, por otro, realizar operaciones SQL en nuestra base de datos mediante el paquete *crate*. La aplicación quedará estará almacenada en el archivo *app.py* que se muestra a continuación:

```
from crate import client
from flask import Flask, request, jsonify
from flask_cors import CORS
import time

app = Flask(__name__)
```

```
CORS(app)

@app.route('/retrievedata',methods=['GET'])
def retrieve_data():
    connection =
        client.connect("http://10.5.0.4:4200/",
        username="crate")

    if connection:
        cursor = connection.cursor()
        cursor.execute("""select table_name
            from information_schema.tables WHERE
            table_name = 'tasks' limit 100""")
            exists = cursor.fetchone()

        if (not exists):
            cursor.execute("""CREATE TABLE tasks
                (name text, task text, date text)""")
            cursor.execute("""COPY tasks FROM
                'file:///db-data/tasks_1_.json'""")
            cursor.execute("""COPY tasks FROM
                'file:///db-data/tasks_2_.json'""")
            time.sleep(1)

        cursor.execute("SELECT * FROM tasks")
        tasks = cursor.fetchall()
        connection.close()

    retorno = {"lTareas": []}
    for t in tasks:
        retorno["lTareas"]
            .append({"name": t[0],
            "task": t[1], "date": t[2]})

    return jsonify(retorno)

@app.route('/newentry', methods=['POST'])
def new_entry():
```

```
    name = request.form['name']
    task = request.form['task']
    date = request.form['date']

    connection =
        client.connect("http://10.5.0.4:4200/",
        username="crate")
    cursor = connection.cursor()
    cursor.execute("""INSERT INTO "doc"."tasks"
        (name, task, date) VALUES (?, ?, ?)""",
        (name, task, date))
    connection.close()

    return "Succesfull"

app.run(host='0.0.0.0', port=4201)
```

Listing 1: Código de app.py

La aplicación cuenta con dos peticiones diferentes, una con método GET y otra con POST con diferentes funcionalidades:

- /retrievedata: se ejecuta cuando carga el frontend. Crea la tabla en la base de datos en caso de que no exista y carga los datos almacenados en el volumen. A continuación, se realiza una operación de SELECT mediante la cual, se recuperan los datos que después se mostrarán en el frontend.
- /newentry: recupera los datos del frontend y, mediante una operación de INSERT, los añade a la base de datos.

Para encapsular la aplicación cliente se ha creado el siguiente Dockerfile, cuyo repositorio en Docker Hub es `https://hub.docker.com/r/srjowy/cliente-crate`:

```
FROM alpine

RUN apk -qq update
RUN apk -qq add python3
RUN apk -qq add py3-pip
RUN pip3 -qq install crate
```

```
RUN pip3 -qq install flask
RUN pip3 -qq install flask_cors
RUN pip3 -qq install docker
RUN mkdir /cliente

COPY ./app.py /cliente/app.py

WORKDIR /cliente

CMD python3 app.py
```

Listing 2: Archivo Dockerfile

En primer lugar, hemos utilizado alpine como imagen base debido a su ligereza. A continuación, instalamos las dependencias necesarias tanto con apk como con pip para las librerías que utilizamos en la aplicación cliente. Después creamos el directorio */cliente* donde almacenamos el archivo python que después ejecutamos para echar a correr la aplicación y que se quede escuchando para tratar las peticiones que le lleguen.

2.3 Frontend

La tercera aplicación implementada es un servidor web. Para implementarlo hemos utilizado la imagen docker *httpd*. Esta aplicación cuenta con un sencillo archivo html que cuenta con tres etiquetas de tipo input que permiten escribir y cuya información se procesará en un archivo javascript que realizará las pertinentes peticiones al backend (aplicación cliente) para almacenar o mostrar la información.

3 Archivo docker-compose

Para lograr que la aplicación tenga sentido en su conjunto, todos los contenedores deben comunicarse entre si. Para ello, se ha creado un archivo docker-compose que gestionará la red y los volúmenes necesarios para el correcto funcionamiento de la base de datos y el frontend.

```
services:
  servidor-bd:
    image: crate
```

```
    ports:
      - 4200:4200
    volumes:
      - ./db:/db-data
    command: ["crate",
              "-Cdiscovery.type=single-node",
              "-Cnetwork.host=_site_"]
    hostname: crate-db
    networks:
      my-network:
        ipv4_address: 10.5.0.4
  cliente-crate:
    build: .
    ports:
      - 4201:4201
    networks:
      my-network:
        ipv4_address: 10.5.0.5
    depends_on:
      - servidor-bd
  web:
    image: httpd
    volumes:
      - ./public:/usr/local/apache2/htdocs
    ports:
      - 8080:80
    networks:
      my-network:
        ipv4_address: 10.5.0.6
    depends_on:
      - cliente-crate
networks:
  my-network:
    driver: bridge
    ipam:
      config:
        - subnet: 10.5.0.0/16
          gateway: 10.5.0.1
```

```
volumes:
  db-data:
```

Listing 3: Archivo docker-compose.yml

El archivo docker-compose cuenta con tres servicios que representan las tres partes de la aplicación. Cada uno tiene una dirección ipv4 que le permitirá identificarse dentro de la red *my-network*.

Por otro lado, cada servicio cuenta con sus dependencias para que todo se ponga en marcha a la vez y sin problemas de conexiones.

En cuanto a la persistencia, se ha creado un volumen al que hemos llamado *db-data* en el que almacenaremos las entradas de la base de datos. Además, copiaremos los datos de la carpeta public a la carpeta htdocs del contenedor web para que tenga disponible los archivos html y javascript del frontend.

4 Kubernetes

Para realizar la implementación de Kubernetes debemos tener en cuenta los objetos necesarios para que la aplicación se ejecute de la misma forma que lo hace con el archivo docker-compose pero con los beneficios de utilizar Kubernetes. Para cada aplicación hemos creado un objeto deployment, con su determinado cluster-ip para que cada contenedor sea visible dentro del cluster y para los contenedores que necesitan persistencia, persistentVolumeClaims. Finalmente, para la conexión con el exterior, hemos creado un objeto de tipo ingress.

Para probar el cluster con todos los objetos se ha utilizado Minikube, que es una manera muy sencilla y rápida para desarrollo de kubernetes.

4.1 Descripción de los objetos kubernetes

- dep_crate: es el objeto deployment que despliega un pod con la imagen de kubernetes. En este caso, al contar CrateDB con la función de crear nodos independientes, la función de réplica resulta muy útil. Además, cuenta con la persistencia que le otorga el persistentVolumeClaim data-crate-vp.

- dep_backend: es el objeto deployment que despliega la imagen que se encuentra en docker-hub. Se pone en marcha y se queda esperando a que le lleguen peticiones.
- dep_web: es el objeto deployment que se encarga de desplegar la imagen httpd. Cuenta con persistencia mediante el volumen data-volume-vp.
- cluster-ip-crate: Otorga una ip dentro del clúster y abre el puerto 4200.
- cluster-ip-backend: Otorga una ip dentro del clúster y abre el puerto 4201.
- cluster-ip-web: Otorga una ip dentro del clúster y abrimos el puerto 8080 que redirige el tráfico al puerto 80.
- ingress: es el objeto ingress que permite que podamos acceder al interior del cluster. En este caso, únicamente nos interesa dar acceso al puerto 8080 que es el puerto abierto para httpd.