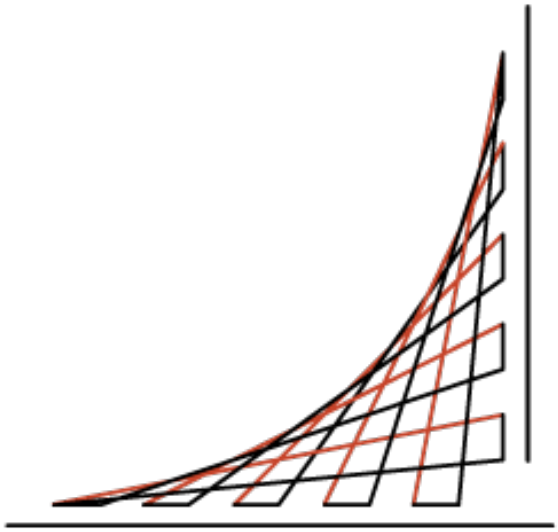

INGENIERÍA DE SISTEMAS - AREP 2020-2

Arquitecturas Empresariales

LABORATORIO 4 : Virtualización: Docker

Luis Daniel Benavides Navarro

Authors:
Juan Camilo Rojas Ortiz

Índice

Resumen

Este artículo presenta la descripción de la arquitectura utilizada para la creación de un sistema construido a partir de contenedores de Docker que permiten modularizar la infraestructura, al asignar maquinas virtuales a cada componente del sistema, de modo que se asegura que cada componente cuenta con los recursos que necesita para su ejecución. El sistema esta compuesto por 5 componentes principales, 1 balanceador de carga que utiliza round robin para distribuir la carga, 3 nodos de la aplicación web cada uno permite el almacenamiento y la consulta de mensajes en una base de datos mongo correspondiente al último componente del sistema. Tanto el balanceador de carga como los nodos de aplicación fueron implementados utilizando el framework Spark. La aplicación fue desplegada sobre una máquina ec2 de AWS.

1. Introducción

Actualmente la infraestructura en la nube ha tomado gran fuerza dado que permite acceder a recursos computacionales muy potentes a precios accesibles y permitiendo que el usuario se evite los esfuerzos de instalación, mantenimiento y compra de estos equipos. Sin embargo conforme crecen los sistemas estos se vuelven complejos de entender y administrar, una forma de manejar esto es por medio de la modularización, por lo que se utiliza la virtualización como una forma de modularizar la infraestructura,una herramienta que permite hacer esto es Docker, que utiliza contenedores para encapsular el software en máquinas virtuales que permiten incluir y aislar todos los recursos que este necesita para funcionar correctamente.

El objetivo de este laboratorio es construir una aplicación de publicación de mensajes que use 1 balanceador de carga, 3 nodos de lógica web y 1 base de datos mongo, teniendo en cuenta que cada uno de estos componentes corresponde a un contenedor Docker. Finalmente todo el sistema debe desplegarse sobre una máquina virtual ec2 de AWS

Para explicar la arquitectura del sistema se explicará el funcionamiento del balanceador de carga ,la base de datos y el nodo web. Además se mostrará la configuración utilizada en los contenedores y en el archivo docker-compose,finalmente el comando utilizado para construir las imágenes y los contenedores.

2. Arquitectura general

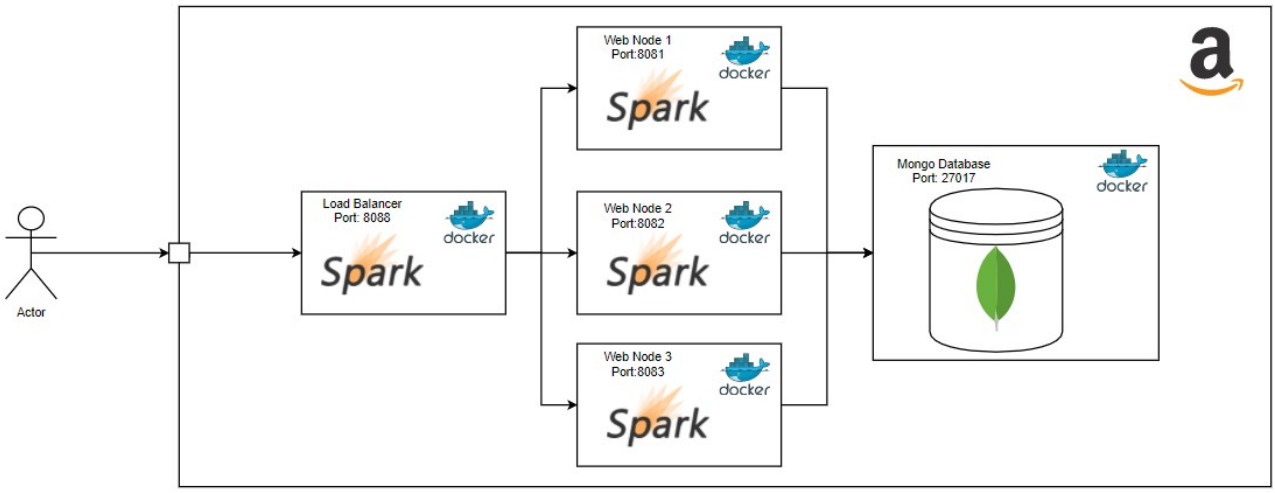

Figura 1: Arquitectura general

En la figura 1 podemos ver la arquitectura general del sistema,el usuario se comunica con el balanceador de carga por medio del puerto 8088, único puerto abierto en la máquina virtual de aws, este balanceador de carga se comunica con 3 nodos de lógica web en los puertos 8081,8082 y 8083, cada balanceador tiene una conexión a una base de datos mongo ubicada en el puerto 27017. Es importante resaltar que cada uno de los componentes descritos anteriormente se encuentra en un contenedor independiente.

2.1. Balanceador de carga

El balanceador de carga esta implementado usando el framework Spark,proporciona dos endpoints GET /mensaje y POST /mensaje, cada uno de estos se encarga de redirigir la petición hacia uno de los nodos de lógica web y propagar la respuesta de estos nodos hacia el usuario, para implementar el envío de peticiones hacia los otros nodos se utilizó la dependencia Okhttp. Para distribuir la carga entre los nodos utiliza el algoritmo de round robin.

```yaml
loadbalancer:
  build:
    context: ./LoadBalancer
    dockerfile: Dockerfile
  depends_on:
    - web
  container_name: loadbalancer

  ports:
  - "8088:6001"
```

Figura 2: Configuración en docker-compose

```
FROM openjdk:8

WORKDIR /loadbalancer/bin

ENV PORT 6001

COPY /target/classes /loadbalancer/bin/classes
COPY /target/dependency /loadbalancer/bin/dependency

CMD ["java","-cp","./classes:./dependency/*","edu.escuelaing.arep.loadbalancer.App"]
```

Figura 3: Configuración en Dockerfile

En la figura 2 podemos ver la configuración del balanceador de carga en el archivo docker-compose, donde se aprecia que se utilizará el puerto de la máquina 8088 para dirigir y recibir las entradas y salidas del puerto 6001 del contenedor, mientras que en la figura 3 se encuentra la configuración del archivo LoadBalancer/Dockerfile, el cual muestra que el balanceador de carga utilizará el puerto 6001 del contenedor

2.2. Nodo Web

La arquitectura de cada nodo se puede observar en la figura 4 y consta de un controlador que publica los endpoints GET /mensajes y POST /mensajes usando el framework Spark, estos endpoints se encargan de enviar al cliente los mensajes que se han almacenado hasta ahora y permitirle publicar un nuevo mensaje respectivamente. Para poder realizar esto el controlador usa una capa de servicios que a su vez se apoya en una capa de persistencia,que es implementada por una conexión a una base de datos mongoDB que permite almacenar los mensajes publicados por medio de documentos, cada

documento consta de un id, la fecha y el contenido del mensaje. Para comunicar las distintas capas se utilizó el principio de inversión de dependencias como se puede ver en la figura 4

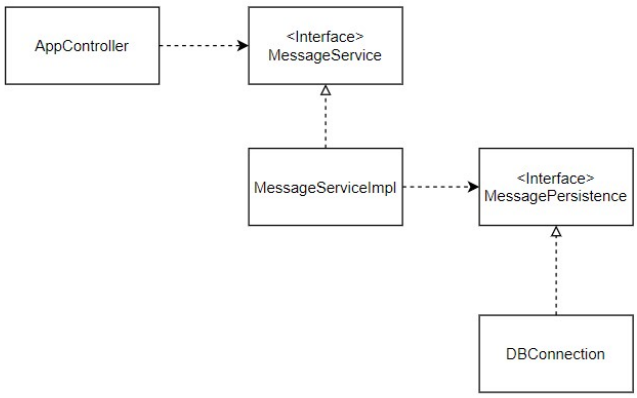

Figura 4: Arquitectura interna

```yaml
web:
  build:
    context: ./App
    dockerfile: Dockerfile
  depends_on:
    - db

  ports:
  - "8081-8083:6000"
```

Figura 5: Configuración nodo docker-compose

```
FROM openjdk:8

WORKDIR /usrapp/bin

ENV PORT 6000
ENV TZ America/Bogota

COPY /target/classes /usrapp/bin/classes
COPY /target/dependency /usrapp/bin/dependency

CMD ["java","-cp","./classes:./dependency/*","edu.escuelaing.arep.app.controllers.App"]
```

Figura 6: Configuración nodo Dockerfile

En la figura 5 podemos ver la configuración de los nodos web en el archivo docker-compose, donde se aprecia que se utilizarán los puertos de la máquina 8081,8082,8083 para dirigir y recibir las entradas y salidas del puerto 6000 de los contenedores generados, por lo que se podrán generar hasta 3 instancias de este nodo, mientras que en la figura 6 se encuentra la configuración del archivo App/Dockerfile, el cual muestra que cada nodo utilizará el puerto 6000 del contenedor

2.3. Base de datos Mongo

Para la creación del servidor de base de datos se utilizó la imagen base de mongo, y sobre esta se creó la base de datos Arep y dentro de esta la colección Mensajes.Para la creación de esta base de datos fue necesaria la creación de un archivo .js encargado de crear un usuario para esta base de datos, con los permisos necesarios, para esto se utilizó el espacio *docker-entrypoint-initdb.d* donde se cargan scripts que se ejecutaran al inicializar la base de datos, la configuración en el archivo docker-compose se puede ver en la figura 7 donde vemos la creación de un usuario administrador y de la base de datos, adicionalmente en la figura 8 se observa el archivo de inicialización .js utilizado para crear el usuario de esa base de datos Arep.

```yaml
  db:
    image: mongo:latest
    container_name: db
    environment:
      MONGO_INITDB_DATABASE: Arep
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: admin
    volumes:
    - ./init-mongo.js:/docker-entrypoint-initdb.d/init-mongo.js:ro
    - mongodb:/data/db
    - mongodb_config:/data/configdb
    ports:
    - 27017:27017
    command: mongod
volumes:
  mongodb:
  mongodb_config:
```

Figura 7: Configuración base de datos mongo docker-compose

```javascript
db.createUser(
    {
        user:"Camilo",
        pwd:"password",
        roles:[{
            role:"readWrite",
            db:"Arep"
        }]
    }
)
```

Figura 8: Script de inicialización init-mongo.js

3. Pruebas

Usando la aplicación localmente desde Docker ToolBox, se inicializa utilizando el comando *docker-compose up -d –scale web=3* y es accesible desde la direccion http://192.168.99.100:8088 (IP asignada por ToolBox)

Figura 9: Publicando mensajes desde el browser a servicio local

Figura 10: Logs de LB, demostrando round robin

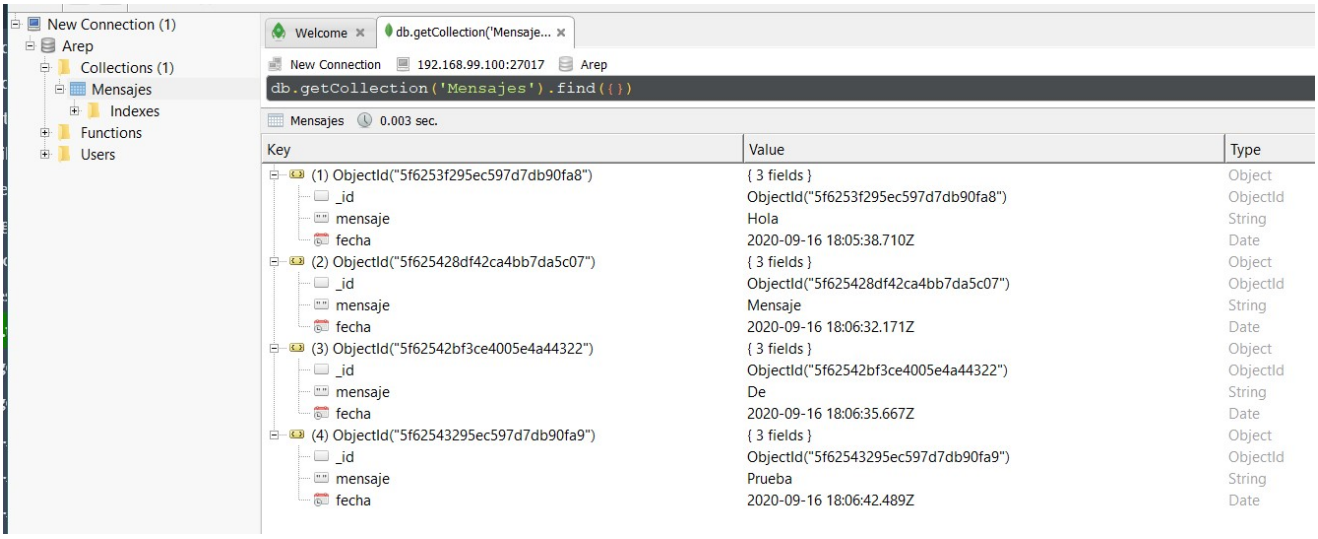

Figura 11: Resultado en base de datos

Se tienen las siguientes pruebas del funcionamiento de la aplicacion sobre AWS

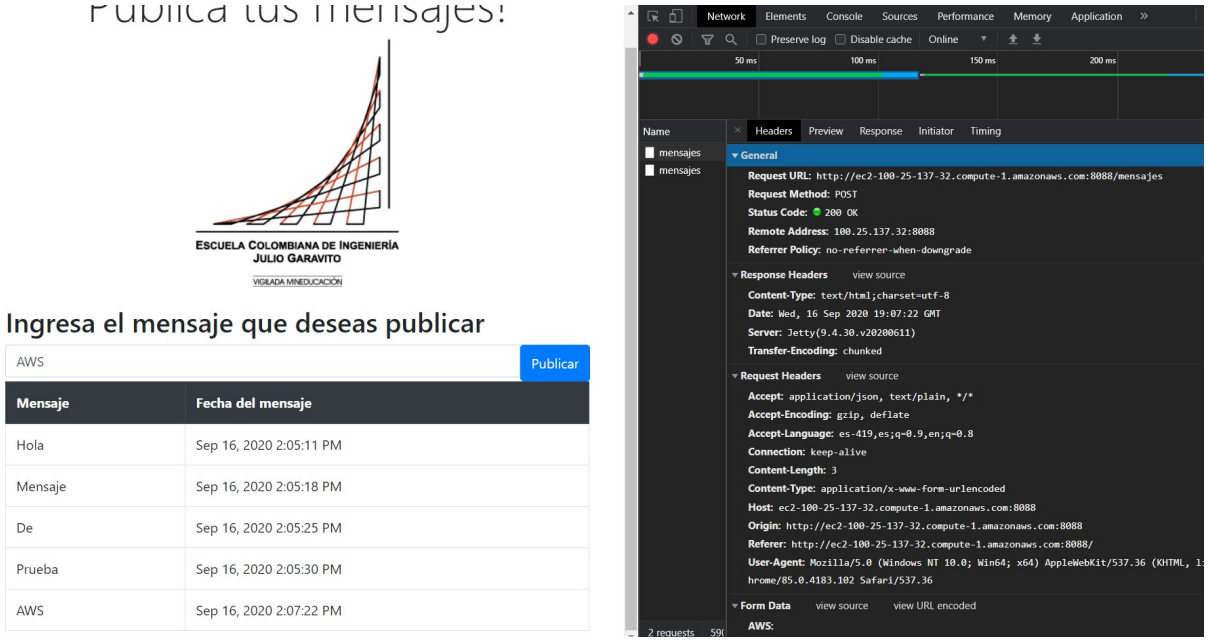

Figura 12: Enviando peticiones al balanceador de carga en AWS

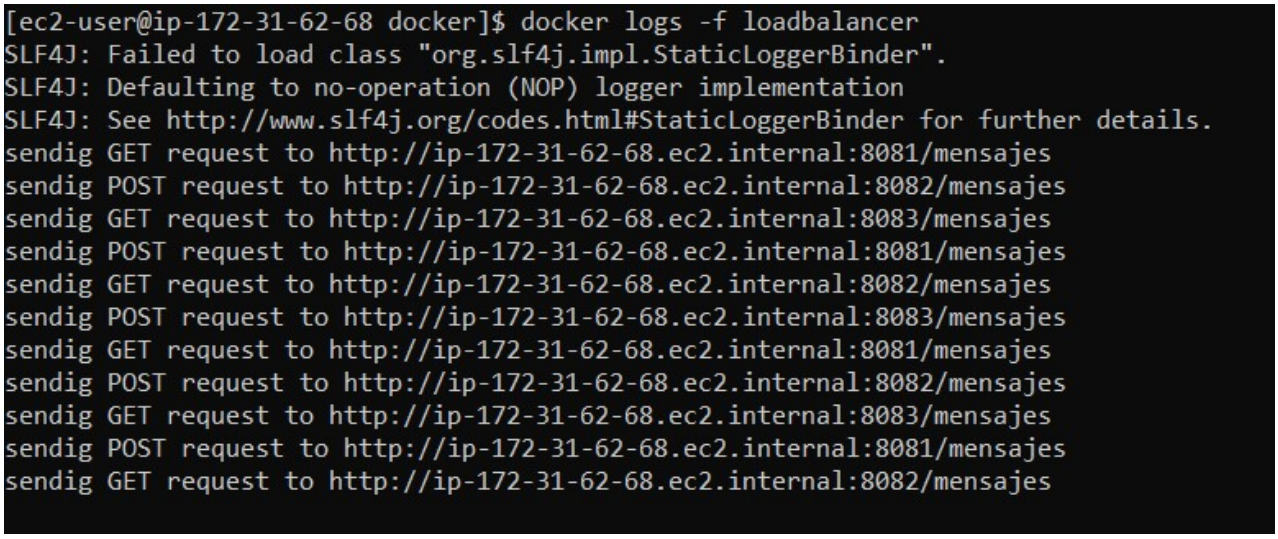

Figura 13: Registro de balanceo de carga en AWS

4. Resultado

Se construyó una aplicacion de publicación de mensajes utilizando una arquitectura de varios componentes que fueron virtualizados en contenedores usando Docker, se implementó un balanceador de carga capaz de distribuir las peticiones sobre 3 nodos de lógica conectados con una base de datos MongoDB.

5. Referencias