

Práctica 7

Computación en la Nube

Pierre Simon Callist Yannick Tondreau



Ejercicio 1

Haciendo uso de la documentación proporcionada en la sección Aplicaciones Cloud Native. Elabora un resumen (una página) en la que defines el concepto de Aplicación Cloud Native

El término de Aplicaciones Cloud Native se refiere a la forma en la que se crean las aplicaciones. Este tipo de aplicaciones, se forman como un conjunto de servicios que son independientes, por lo tanto la aplicación puede estar separada en diferentes partes.

El uso de esta arquitectura para el diseño de aplicaciones puede suponer las siguientes ventajas:

- **Escalabilidad:** Mediante el uso de tecnologías como la contenerización, el despliegue en horizontal se realiza de una forma muy sencilla
- **Velocidad de Despliegue:** Integrándose cada vez más estrategias de CI/CD, las empresas pueden desplegar rápidamente un cambio en desarrollo a la aplicación final.

Por lo tanto, podemos concluir que las Aplicaciones Cloud Native, están diseñadas para que los desarrolladores puedan desplegar más rápidamente, de esta forma permitiendo realizar cambios que requieren los usuarios de forma más rápida. También permite que cuando la carga de trabajo sea mayor, sea posible escalar de forma sencilla los servicios en ejecución. Estos aspectos son muy importantes, dado que actualmente las empresas y el mercado se mueven muy rápidamente y las aplicaciones deben de estar preparadas para dichos cambios.

Además, gracias a grandes empresas como Google y Amazon, el despliegue de servicios en la nube se ha convertido en una tarea fácil de realizar, ofreciendo precios muy asequibles que se basan en el uso de la infraestructura ofrecida.



Ejercicio 2

El documento *Chapter 3. Designing Cloud Native Applications* recoge las características fundamentales para el diseño de una aplicación Cloud Native. Describe y documenta el diseño de la aplicación que permite realizar el procesamiento de una imagen en paralelo de manera que pueda considerarse una aplicación Cloud Native. Incorpora en el diseño los elementos que necesites incorporar del framework que has venido desarrollando en prácticas anteriores.

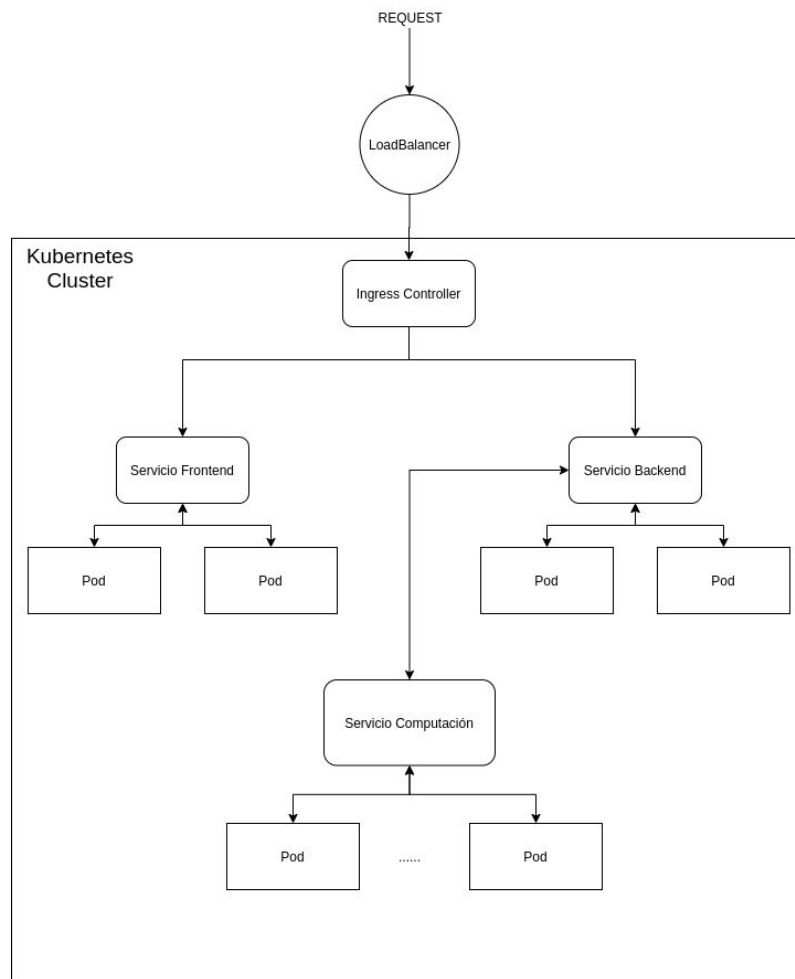
El documento *Chapter 3. Designing Cloud Native Applications* nos presenta las características de una aplicación Cloud Native, dichas características se reparten en los siguientes puntos:

- **Operational Excellence:** se basa en cómo ejecutar, monitorizar y mejorar la aplicación a medida que avanza el tiempo y la forma de implementarlo es usando técnicas de DevOps.
 - Automatizar lo máximo posible
 - Monitorizar todo
 - Documentar todo
 - Realizar cambios incrementales
 - Diseño para fallos
- **Security:** el aspecto de la seguridad en los proveedores cloud está bastante controlado, dado que constan de un equipo de expertos que se aseguran de que la infraestructura sea segura, aun así, debemos de asegurarnos que la aplicación desarrollada también sea segura.
- **Reliability and Availability:** con fiabilidad se refiere a que la aplicación funcione aún ante la presencia de fallos, y con disponibilidad se refiere a que la aplicación esté disponible siempre que un usuario quiera acceder a ella.
- **Scalability and Cost:** estos dos conceptos van de la mano, cuando se diseña una aplicación Cloud Native no solo se debe de pensar en cómo escalar la aplicación en el caso de necesitarlo, si no también como realizarlo de una forma que sea rentable.



Teniendo en consideración los puntos anteriormente mencionados, el diseño resultante del framework desarrollado es el siguiente.

Diseño de la Aplicación



Para el diseño de la aplicación se ha decidido usar *Kubernetes*. Según su propia definición, *Kubernetes* es un sistema de orquestación de contenedores open-source para automatizar la implementación, el escalado y la administración de aplicaciones. Según esta definición puede parecer evidente que *Kubernetes* sea una de las tecnologías más usadas para el desarrollo y despliegue de aplicaciones Cloud Native.

A continuación se explicará más en detalle cada uno de los elementos que componen el diseño.



Cluster de Kubernetes

Como lo indica su nombre, se trata de un cluster de varias máquinas que ejecuta Kubernetes. En él, se encontrarán los diferentes elementos que podemos lanzar en Kubernetes, como lo son los Pods, los ReplicaSet, los Deployments, los Servicios, los Ingress y el IngressController.

Al definir los elementos anteriores de Kubernetes, este se ocupará de controlar el estado de los Pods, el tráfico de la Red, etc. Por lo tanto, no debemos preocuparnos de si un Pod falla, puesto que si esto pasa, Kubernetes se encargará de lanzar un nuevo Pod para nuestro Servicio.

Ingress Controller

Este elemento permite acceso a los Ingress que se definan (en la imagen no se muestra, pero existe un Ingress para el Servicio Backend y otro para el Frontend) en el cluster.

Los Ingress son objetos API que maneja el acceso externo a los servicios del cluster. Proporciona balanceo de carga, definición de rutas HTTP y HTTPS, terminación SSL y definir nombres para cada Servicio.

Además, el uso de Ingress e IngressController, permite ahorrar costes dado que de esta forma no es necesario asignar una IP pública a cada Servicio, se define una ruta para cada uno de los Servicios y se accede desde el balanceador de carga definido por el proveedor de infraestructura cloud.

Servicios

En el cluster de Kubernetes diseñado se lanzarán los siguientes servicios:

- **Backend:** servicio que ejecutará el servidor backend que se ha desarrollado. Dispone de dos Pods para mantener un nivel de disponibilidad mayor en caso de fallo de un Pod.
- **Frontend:** servicio que ejecutará el servidor frontend que se ha desarrollado en las prácticas. Dispone de dos Pods para mantener un nivel de disponibilidad mayor en caso de fallo de un Pod.
- **Computación:** servicio en el cual se lanzarán los programas que vayan definiendo los usuarios en la aplicación final. El número de pods que se le define puede ser variable dado que simula el cluster externo que se ha desarrollado en la práctica 2.



Ejercicio 3

Instala el entorno Apache Spark y ejecuta algún ejemplo sencillo en python.

Con el fin de familiarizarme con el entorno de Spark, he realizado la instalación de dos formas distintas.

Docker - Jupyter Notebook

Como prueba inicial para familiarizarse con el entorno de Spark, se ha decidido probarlo mediante un contenedor de Docker que también contiene un Jupyter Notebook para poder usarlo. La forma de ejecutar el contenedor es de la siguiente forma:

```
docker run -d -p 80:8888 -v $PWD:/home/jovyan/work --name spark jupyter/pyspark-notebook
```

Una vez en ejecución el contenedor, navegamos a <http://localhost> y obtenemos el siguiente resultado.

The screenshot shows the Jupyter Notebook web interface. At the top, there's a navigation bar with the Jupyter logo and 'Quit' and 'Logout' buttons. Below this is a tabbed interface with 'Files', 'Running', and 'Clusters'. The 'Files' tab is selected, displaying a file browser for the '/ work' directory. It includes a search bar, 'Upload', 'New', and 'Refresh' buttons. The file list shows: a '..' directory entry, a 'data' directory, a 'Testing.ipynb' file which is currently 'Running', and a 'GlobalLandTemperaturesByCountry.csv' file. Each entry shows its last modified time and file size.

Name	Last Modified	File size
..	seconds ago	
data	4 minutes ago	
Testing.ipynb	seconds ago	2.06 kB
GlobalLandTemperaturesByCountry.csv	2 days ago	22.7 MB

Creamos un Jupyter Notebook y ejecutamos el siguiente ejemplo que nos ofrece Spark desde su repositorio de [Github](https://github.com). El resultado obtenido es el que se muestra en la imagen siguiente.



```
jupyter Testing Last Checkpoint: a few seconds ago (autosaved) Python 3 Logout
File Edit View Insert Cell Kernel Widgets Help
In [4]: from pyspark.ml.clustering import KMeans
        from pyspark.ml.evaluation import ClusteringEvaluator
        from pyspark.sql import SparkSession

In [5]: spark = SparkSession\
        .builder\
        .appName("KMeansExample")\
        .getOrCreate()

        # $example on$
        # Loads data.
        dataset = spark.read.format("libsvm").load("data/sample_k_means.txt")

        # Trains a k-means model.
        kmeans = KMeans().setK(2).setSeed(1)
        model = kmeans.fit(dataset)

        # Make predictions
        predictions = model.transform(dataset)

        # Evaluate clustering by computing Silhouette score
        evaluator = ClusteringEvaluator()

        silhouette = evaluator.evaluate(predictions)
        print("Silhouette with squared euclidean distance = " + str(silhouette))

        # Shows the result.
        centers = model.clusterCenters()
        print("Cluster Centers: ")
        for center in centers:
            print(center)
        # $example off$

        spark.stop()

Silhouette with squared euclidean distance = 0.9997530305375207
Cluster Centers:
[0.1 0.1 0.1]
[9.1 9.1 9.1]
```

Cluster Spark en Docker

Ahora que hemos podido comprobar el funcionamiento de Spark, procederemos a ejecutar un cluster de Spark mediante contenedores Docker y le enviaremos el código a ejecutar. Para conseguirlo se ha seguido este [tutorial](#), la diferencia con el tutorial es que usaremos la versión estable más reciente de Spark (2.4.5).

Una vez creada la imagen Docker con nuestro Dockerfile, creamos una red para los contenedores Spark mediante el siguiente comando.

```
docker network create --driver bridge spark-network
```



Seguidamente creamos un contenedor master y dos workers, usando los siguientes comandos respectivamente.


```
docker run -d -t --name master --network spark-network pierresimt/spark
```

```
docker exec master start-master
```

```
docker run -d -t --name worker-X --network spark-network pierresimt/spark
```

```
docker exec worker-X start-slave spark://master:7077
```

El resultado de ejecutar los comandos anteriores y navegando a la IP del contenedor master es el siguiente:

 **Spark Master at spark://1be4edee70ed:7077**

URL: spark://1be4edee70ed:7077
Alive Workers: 2
Cores in use: 16 Total, 0 Used
Memory in use: 29.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (2)

Worker Id	Address	State	Cores	Memory
worker-20200511112754-172.18.0.3-36793	172.18.0.3:36793	ALIVE	8 (0 Used)	14.5 GB (0.0 B Used)
worker-20200511113022-172.18.0.4-41485	172.18.0.4:41485	ALIVE	8 (0 Used)	14.5 GB (0.0 B Used)

Running Applications (0)


Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	----------------	------	-------	----------

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	----------------	------	-------	----------

Para comprobar que todo funciona correctamente, lanzaremos uno de los ejemplos que nos ofree Spark. Para lanzar el ejemplo, lo realizaremos de la siguiente forma:

```
./spark/bin/spark-submit --master spark://172.18.0.2:7077  
spark/examples/src/main/python/pi.py 1000 2>/dev/null
```

 **Spark Master at spark://1be4edee70ed:7077**

URL: spark://1be4edee70ed:7077
Alive Workers: 2
Cores in use: 16 Total, 16 Used
Memory in use: 29.0 GB Total, 2.0 GB Used
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (2)

Worker Id	Address	State	Cores	Memory
worker-20200511112754-172.18.0.3-36793	172.18.0.3:36793	ALIVE	8 (8 Used)	14.5 GB (1024.0 MB Used)
worker-20200511113022-172.18.0.4-41485	172.18.0.4:41485	ALIVE	8 (8 Used)	14.5 GB (1024.0 MB Used)

Running Applications (1)


Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20200511133223-0000	(kill) PythonPi	16	1024.0 MB	2020/05/11 13:32:23	ptondreau	RUNNING	5 s

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	----------------	------	-------	----------



Cuando termine su ejecución finalmente tendremos:

 **Spark Master at spark://1be4edee70ed:7077**

URL: spark://1be4edee70ed:7077
Alive Workers: 2
Cores in use: 16 Total, 0 Used
Memory in use: 29.0 GB Total, 0.0 B Used
Applications: 0 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

↳ Workers (2)

Worker Id	Address	State	Cores	Memory
worker-20200511112754-172.18.0.3-36793	172.18.0.3:36793	ALIVE	8 (0 Used)	14.5 GB (0.0 B Used)
worker-20200511113022-172.18.0.4-41485	172.18.0.4:41485	ALIVE	8 (0 Used)	14.5 GB (0.0 B Used)

↳ Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	----------------	------	-------	----------

↳ Completed Applications (1)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20200511133223-0000	PythonPi	16	1024.0 MB	2020/05/11 13:32:23	ptondreau	FINISHED	41 s

Y finalmente, en la consola obtendremos el resultado del ejemplo ejecutado:

```
root@493f425c3e98:/home# spark-submit --master spark://master:7077 spark/examples/src/main/python/pi.py 1000 2>/dev/null
Pi is roughly 3.141558
```