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

Curso académico 2019/2020

Trabajo de Fin de Máster

Java Kubernetes

Autor: Francisco Javier Avilés López
Tutor: Micael Gallego Carrillo

Índice

Resumen

Hoy en día la gran mayoría de desarrolladores hacen uso de Docker como herramienta para facilitar la ejecución de aplicaciones web desde entornos de desarrollo hasta entornos productivos. Como consecuencia, han surgido diversos orquestadores de imágenes Docker como Kubernetes, con el fin de automatizar el despliegue y manejo de aplicaciones en contenedores. Kubernetes cuenta ya con seis años de madurez desde que fue creado inicialmente por Google, siendo a día de hoy el más utilizado en el sector.

Esto, sumado a que absolutamente todos los principales proveedores cloud (Amazon – EKS, Google GKE, Azure – AKS , etcétera) ofrecen ya un servicio de Kubernetes, hace que cada día más empresas decidan migrar sus aplicaciones a estas plataformas.

Hay muchísimos recursos en internet sobre Kubernetes, pero casi todos tienen un enfoque para principiantes que se quieren iniciar en el uso de la plataforma. Además, no suelen tener un enfoque específico a un lenguaje de programación. Por este motivo, siendo Kubernetes el orquestador de imágenes Docker más utilizado con diferencia, y java uno de los lenguajes en los que más se programan hoy en día aplicaciones web, tiene mucho sentido tratar en profundidad cómo desarrollar aplicaciones Java implementando los principales conceptos de Kubernetes; tratando desde el comportamiento de la JVM en contenedores Docker hasta qué ofrecen los principales frameworks Java con enfoque a cloud y microservicios para implementar estos conceptos que requiere una aplicación “cloud-native” en Kubernetes.

Ya que el enfoque de los análisis será tratar con cierta profundidad cómo implementar en Java varios de estos conceptos de Kubernetes, en todo momento se dará por hecho que la audiencia de los análisis serán desarrolladores con una base de conocimientos tanto en Docker, como en Kubernetes, como en Java.

Durante cada análisis, no solo se expondrán los resultados y evidencias del mismo, si no el cómo llegar a los mismos, proporcionando en todo momento los comandos, explicaciones y código necesarios para seguir los pasos que se llevan a cabo a lo largo del análisis sobre un clúster Kubernetes cualquiera.

La temática concreta de cada análisis se ha escogido de forma que se cubra cómo dar solución a los principales desafíos y problemáticas que un desarrollador puede encontrar a la hora de implementar una aplicación “cloud-native” en Java para ser desplegada en Kubernetes. Además, debido al auge de diversos frameworks como Quarkus, SpringBoot o Micronaut, surge la tesitura de elegir “el mejor” para cada caso, por tanto, los análisis realizados intentan cubrir una misma solución con cada uno de ellos, ofreciendo comparativas y pequeñas reflexiones, con el fin de ayudar al lector a decantarse por uno u otro según sus necesidades.

Por último, mencionar que a pesar de que en esta memoria académica se incluirán los principales detalles de cada análisis que se ha llevado a cabo, describiendo los pasos que se han llevado a cabo y los principales resultados y conclusiones, los análisis en formato markdown, así como todo el código asociado a cada proyecto java ha sido publicado en un repositorio GIT de la universidad (https://github.com/MasterCloudApps-Projects/Java-Kubernetes/).

Introducción y objetivos

A modo de introducción, una pequeña presentación de las tres tecnologías que vertebran este TFM ayudará tanto a comprender la motivación del mismo como los objetivos que persigue.

Docker

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.

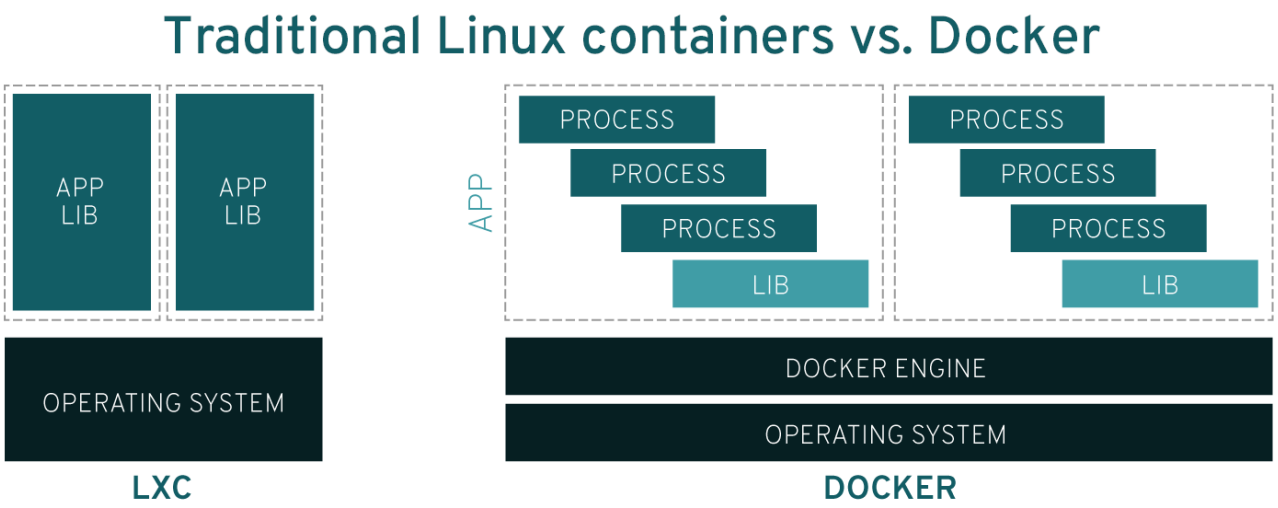
Docker está transformando la forma en que se desarrolla, distribuye y ejecuta el software. La ventaja es muy evidente, podemos encapsular todo el entorno de trabajo de manera que los desarrolladores saben que pueden estar trabajando en su servidor local, con la seguridad de que, al llegar el momento de ponerlo en producción, van a estar ejecutándose con la misma configuración sobre la que se han hecho todas las pruebas.

La tecnología Docker usa el kernel de Linux y las funciones de este, como Cgroups y namespaces, para segregar los procesos, de modo que puedan ejecutarse de manera independiente. El propósito de los contenedores es esta independencia: la capacidad de ejecutar varios procesos y aplicaciones por separado para hacer un mejor uso de su infraestructura y, al mismo tiempo, conservar la seguridad que tendría con sistemas separados.

Las herramientas del contenedor, como Docker, ofrecen un modelo de implementación basado en imágenes. Esto permite compartir una aplicación, o un conjunto de servicios, con todas sus dependencias en varios entornos. Docker también automatiza la implementación de la aplicación (o conjuntos combinados de procesos que constituyen una aplicación) en este entorno de contenedores.

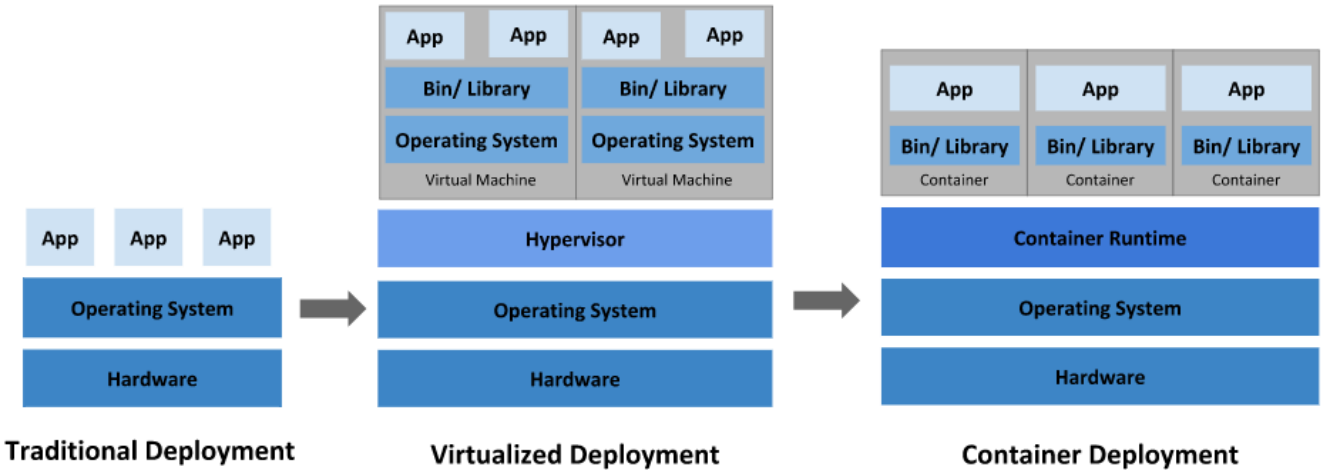
Estas herramientas desarrolladas a partir de los contenedores de Linux, lo que hace a Docker fácil de usar y único, otorgan a los usuarios un acceso sin precedentes a las aplicaciones, la capacidad de implementar rápidamente y control sobre las versiones y su distribución.

Al contrario de lo que se piensa, la tecnología Docker no es la misma que la de los contenedores Linux tradicionales. Al principio, la tecnología Docker se desarrolló a partir de la tecnología LXC, lo que la mayoría de las personas asocia con contenedores de Linux "tradicionales", aunque desde entonces se ha alejado de esa dependencia. LXC era útil como virtualización ligera, pero no ofrecía una buena experiencia al desarrollador ni al usuario. La tecnología Docker no solo aporta la capacidad de ejecutar contenedores; también facilita el proceso de creación y diseño de contenedores, de envío de imágenes y de creación de versiones de imágenes (entre otras cosas).



1. Comparativa tecnología LXC vs Docker

Para entender realmente por qué la aparición de Docker en el mundo del desarrollo ha sido una revolución, es importante echar un ojo a cómo ha evolucionado el despliegue de aplicaciones desde el enfoque tradicional, al enfoque basado en contenedores:



2. Evolución del despliegue de aplicaciones

Implementación tradicional: al principio, las organizaciones ejecutaban aplicaciones en servidores físicos. No había forma de definir límites de recursos para aplicaciones en un servidor físico y esto provocó problemas de asignación de recursos. Por ejemplo, si se ejecutan varias aplicaciones en un servidor físico, puede haber instancias en las que una aplicación consumiría la mayoría de los recursos y, como resultado, las otras aplicaciones tendrían un rendimiento inferior. Una solución para esto sería ejecutar cada aplicación en un servidor físico diferente. Pero esto no proliferó ya que los recursos estaban infrautilizados y era caro para las organizaciones mantener muchos servidores físicos.

Implementación virtualizada: como solución, se introdujo la virtualización. Permite ejecutar múltiples máquinas virtuales (VM) en la CPU de un solo servidor físico. La virtualización permite aislar las aplicaciones entre máquinas virtuales y proporciona un nivel de seguridad, ya que otra aplicación no puede acceder libremente a la información de una aplicación.

La virtualización permite una mejor utilización de los recursos en un servidor físico y permite una mejor escalabilidad porque una aplicación se puede agregar o actualizar fácilmente, reduce los costos de hardware y mucho más. Con la virtualización, puede presentar un conjunto de recursos físicos como un clúster de máquinas virtuales desechables.

Cada máquina virtual es una máquina completa que ejecuta todos los componentes, incluido su propio sistema operativo, además del hardware virtualizado.

Implementación de contenedores: los contenedores son similares a las máquinas virtuales, pero tienen propiedades de aislamiento para compartir el sistema operativo (SO) entre las aplicaciones. Por tanto, los contenedores son muy ligeros. Al igual que una máquina virtual, un contenedor tiene su propio sistema de archivos, CPU, memoria, espacio de proceso y más. Como están desacoplados de la infraestructura subyacente, son portátiles a través de distribuciones de SO y nubes.

Los contenedores se han vuelto populares porque ofrecen diversos beneficios adicionales, como:

- Creación e implementación de aplicaciones de manera ágil: mayor facilidad y eficiencia de la creación de imágenes de contenedores en comparación con el uso de imágenes de VM.
- Desarrollo, integración e implementación continuos: proporciona una construcción e implementación de imágenes de contenedores confiables con reversiones rápidas y fáciles (debido a la inmutabilidad de la imagen).
- Separación de preocupaciones de Dev y Ops: capacidad de creación de imágenes de contenedores de aplicaciones en el momento de la compilación / lanzamiento en lugar de en el momento de la implementación, separando así las aplicaciones de la infraestructura.
- La observabilidad no solo muestra información y métricas a nivel del sistema operativo, sino también el estado de la aplicación y otras métricas.
- Consistencia en los diferentes entornos: funciona de la misma manera en una computadora portátil que en la nube.
- Portabilidad de distribución de SO y nube: se ejecuta en Ubuntu, RHEL, CoreOS, en las instalaciones, en las principales nubes públicas y en cualquier otro lugar.
- Gestión centrada en aplicaciones: aumenta el nivel de abstracción de ejecutar un sistema operativo en hardware virtual a ejecutar una aplicación en un sistema operativo utilizando recursos lógicos.
- Microservicios libremente acoplados, distribuidos, elásticos y liberados: las aplicaciones se dividen en partes más pequeñas e independientes y se pueden implementar y administrar de forma dinámica, no una pila monolítica que se ejecuta en una gran máquina de un solo propósito.
- Aislamiento de recursos: rendimiento de la aplicación predecible.
- Utilización de recursos: alta eficiencia y densidad.

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.

Google liberó el proyecto Kubernetes en el año 2014. Kubernetes se basa en la experiencia de Google corriendo aplicaciones en producción a gran escala por década y media, junto a las mejores ideas y prácticas de la comunidad.

Kubernetes ofrece un entorno de administración centrado en contenedores. orquestando la infraestructura de cómputo, redes y almacenamiento para que las cargas de trabajo de los usuarios no tengan que hacerlo. Esto ofrece la simplicidad de las Plataformas como Servicio (PaaS) con la flexibilidad de la Infraestructura como Servicio (IaaS) y permite la portabilidad entre proveedores de infraestructura.

A pesar de que Kubernetes ya ofrece muchas funcionalidades, siempre hay nuevos escenarios que se benefician de nuevas características. Los flujos de trabajo de las aplicaciones pueden optimizarse para acelerar el tiempo de desarrollo. Una solución de orquestación propia puede ser suficiente al principio, pero suele requerir una automatización robusta cuando necesita escalar. Es por ello que Kubernetes fue diseñada como una plataforma: para poder construir un ecosistema de componentes y herramientas que hacen más fácil el desplegar, escalar y administrar aplicaciones.

Las etiquetas, o Labels, permiten a los usuarios organizar sus recursos como deseen. Las anotaciones, o Annotations, les permiten asignar información arbitraria a un recurso para facilitar sus flujos de trabajo y hacer más fácil a las herramientas administrativas inspeccionar el estado.

Además, el Plano de Control de Kubernetes usa las mismas APIs que usan los desarrolladores y usuarios finales. Los usuarios pueden escribir sus propios controladores, como por ejemplo un planificador o scheduler, usando sus propias APIs desde una herramienta de línea de comandos.

Este diseño ha permitido que otros sistemas sean construidos sobre Kubernetes.

Kubernetes no es una Plataforma como Servicio (PaaS) convencional. Ya que Kubernetes opera a nivel del contenedor y no a nivel del hardware, ofrece algunas características que las PaaS también ofrecen, como deployments, escalado, balanceo de carga, registros y monitorización. Dicho esto, Kubernetes no es monolítico y las soluciones que se ofrecen de forma predeterminada son opcionales e intercambiables.

Por tanto, podemos decir que Kubernetes:

- No limita el tipo de aplicaciones que soporta.
- No hace deployment de código fuente ni compila tu aplicación.
- No provee servicios en capa de aplicación como middleware (por ejemplo, buses de mensaje), frameworks de procesamiento de datos (como Spark), bases de datos (como MySQL), caches o sistemas de almacenamiento (como Ceph).

- No dictamina las soluciones de registros, monitoreo o alerta que se deben usar.
- No provee ni obliga a usar un sistema o lenguaje de configuración (como jsonnet) sino que ofrece una API declarativa que puede ser usada con cualquier forma de especificación declarativa
- No provee ni adopta un sistema exhaustivo de mantenimiento, administración o corrección automática de errores

Además, Kubernetes no es un mero sistema de orquestación. De hecho, Kubernetes elimina la necesidad de orquestar. Orquestación se define como la ejecución de un flujo de trabajo definido: haz A, luego B y entonces C. Kubernetes está compuesto de un conjunto de procesos de control independientes y combinables entre sí que llevan el estado actual hacia el estado deseado. No debería importar demasiado como llegar de A a C. No se requiere control centralizado y, como resultado, el sistema es más fácil de usar, más poderoso, robusto, resiliente y extensible.

Desde el punto de vista del desarrollador, es interesante crear aplicaciones nativas de la nube con Kubernetes como plataforma de tiempo de ejecución mediante el uso de patrones de Kubernetes. Los patrones son las herramientas que necesita un desarrollador de Kubernetes para crear aplicaciones y servicios basados en contenedores.

Con Kubernetes puedes:

- Organizar contenedores en varios hosts.
- Hacer un mejor uso del hardware para maximizar los recursos necesarios para ejecutar aplicaciones
- Controlar y automatizar las implementaciones y actualizaciones de aplicaciones.
- Montar y agregar almacenamiento para ejecutar aplicaciones con estado.
- Escalar aplicaciones en contenedores y sus recursos sobre la marcha.
- Administrar los servicios de manera declarativa, lo que garantiza que las aplicaciones implementadas siempre se estén ejecutando de la manera que se requiere.
- Comprobar el estado y autocorrección de las aplicaciones de forma automática, reinicio automático, replicación automática y ajuste de escala automático.

Como ocurre con la mayoría de las tecnologías, el lenguaje específico de Kubernetes puede actuar como una barrera de entrada. Analicemos algunos de los términos más comunes para ayudar a comprender mejor Kubernetes.

Plano de control: la colección de procesos que controlan los nodos de Kubernetes. Aquí es donde se originan todas las asignaciones de tareas.

Nodos: estas máquinas realizan las tareas solicitadas asignadas por el plano de control.

Pod: un grupo de uno o más contenedores implementados en un solo nodo. Todos los contenedores de un pod comparten una dirección IP, IPC, nombre de host y otros recursos. Los pods abstraen la red y el

almacenamiento del contenedor subyacente. Esto permite mover contenedores por el clúster más fácilmente.

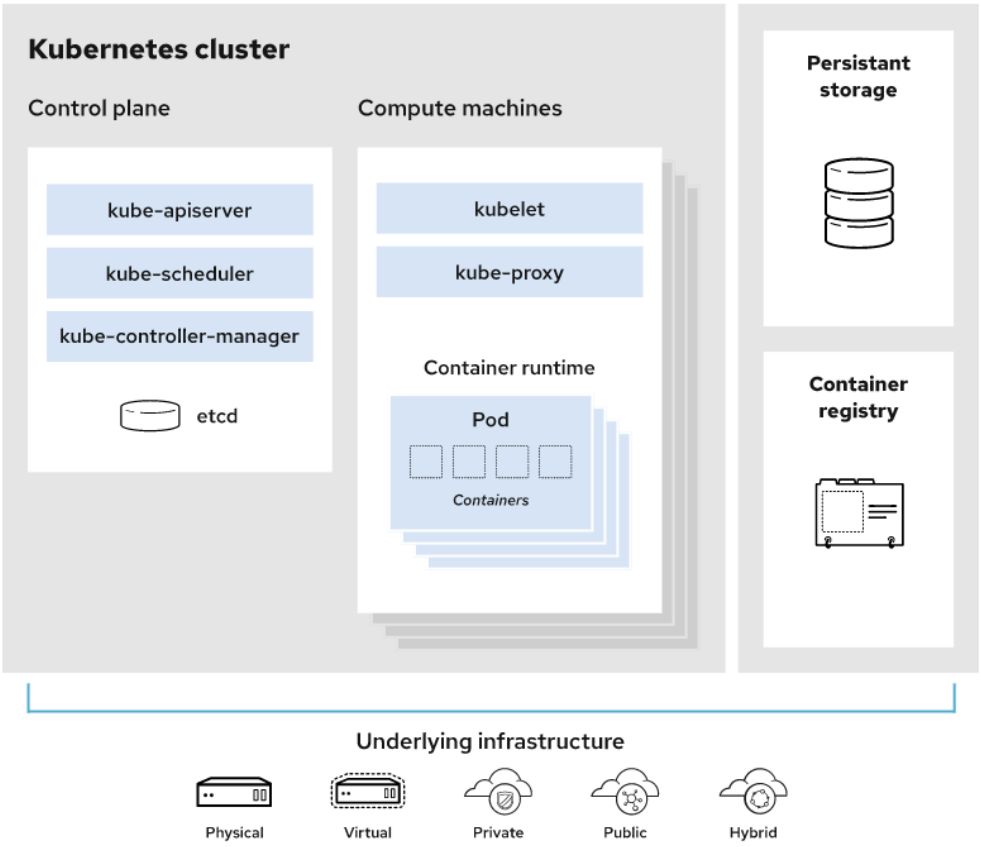
Controlador de replicación: controla cuántas copias idénticas de un pod deberían ejecutarse en algún lugar del clúster.

Servicio: separa las definiciones de tareas de los pods. Los proxies de servicios de Kubernetes envían automáticamente las solicitudes de servicio al pod correspondiente, sin importar adónde se traslada en el clúster, o incluso si está siendo reemplazado.

Kubelet: este servicio se ejecuta en los nodos y lee los manifiestos del contenedor, y garantiza que los contenedores definidos estén iniciados y ejecutándose.

kubectl: la herramienta de configuración de línea de comandos para Kubernetes.

Pero, ¿cómo funciona exactamente Kubernetes?



3. Visión general de Kubernetes

Una implementación de Kubernetes en funcionamiento se denomina clúster. Se puede visualizar un clúster de Kubernetes como dos partes: el plano de control y las máquinas o nodos de cómputo.

Cada nodo tiene su propio entorno Linux y puede ser una máquina física o virtual. Cada nodo ejecuta pods, que se componen de contenedores.

El plano de control es responsable de mantener el estado deseado del clúster, como qué aplicaciones se están ejecutando y qué imágenes de contenedor utilizan. Las máquinas informáticas realmente ejecutan las aplicaciones y las cargas de trabajo.

Kubernetes se ejecuta sobre un sistema operativo e interactúa con pods de contenedores que se ejecutan en los nodos.

El plano de control de Kubernetes toma los comandos de un administrador (o equipo de DevOps) y transmite esas instrucciones a las máquinas.

Esta transferencia funciona con una multitud de servicios para decidir automáticamente qué nodo es el más adecuado para la tarea. Luego, asigna recursos y asigna los pods en ese nodo para cumplir con el trabajo solicitado.

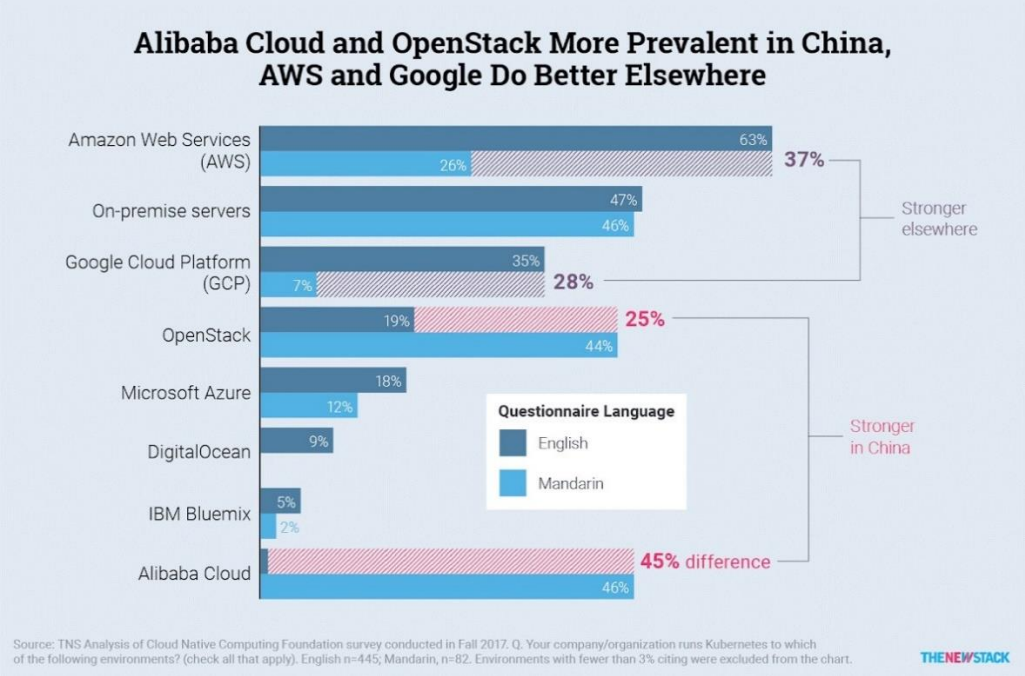
El estado deseado de un clúster de Kubernetes define qué aplicaciones u otras cargas de trabajo deben ejecutarse, junto con las imágenes que usan, los recursos que deben estar disponibles para ellos y otros detalles de configuración similares.

Desde el punto de vista de la infraestructura, hay pocos cambios en la forma de administrar los contenedores. Su control sobre los contenedores simplemente ocurre en un nivel superior, lo que le proporciona un mejor control sin la necesidad de microgestionar cada contenedor o nodo por separado.

Su trabajo implica configurar Kubernetes y definir nodos, pods y los contenedores dentro de ellos. Kubernetes maneja la orquestación de los contenedores.

El lugar donde se ejecute Kubernetes es muy variable. Puede ser en máquinas virtuales, proveedores de nube pública, nubes privadas y entornos de nube híbrida. Una de las ventajas clave de Kubernetes es que funciona en muchos tipos diferentes de infraestructura.

En una encuesta en thenewstack.io, se muestra en un gráfico el uso de proveedores según localización geográfica.



4. Encuesta tipo de proveedor cloud

Java

En el 2020 Java sigue siendo uno de los lenguajes de programación más utilizados a la hora de desarrollar aplicaciones web, gozando de un ciclo de release con muy buena cadencia (seis meses, va por la versión 15) y un rendimiento muy optimizado.

Pese a los esfuerzos por JavaEE/JakartaEE de adaptarse al movimiento cloud con el estándar Microprofile y sus diversas implementaciones (Payara micro, Openliberty, Thorntail…), otros frameworks parecen haber cogido la delantera en esta carrera, como son SpringBoot, Micronaut y Quarkus. El motivo principal es la rápida adaptación que tienen a las necesidades que van surgiendo en la indsutria, bien sea a la hora de tener una extensión de kubernetes para facilitar la implementación de una aplicación cloud-native, para generar unos boilerplates livianos orientados a microservicios o para correr la aplicación sobre GraalVM en vez de sobre la JVM, por ejemplo, para serverless o autoscaling a 0.

Los tres frameworks se pueden ejecutar con Maven. Algo que tienen también en común es que todos ellos cuentan con una web muy similar donde se pueden generar los correspondientes boilerplates, seleccionando la paquetización y las dependencias con las que el proyecto se quiere inicializar:

- Quarkus: https://code.quarkus.io/
- Micronaut: https://micronaut.io/launch/
- SpringBoot: https://start.spring.io/

Una vez se descargan los proyectos de las correspondientes webs, y teniendo maven instalado, ya serían proyectos funcionales con las dependencias seleccionadas que podríamos ejecutar. Además, todos ellos cuentan con modo desarrollo permitiendo “hot swap” (y funciona muy bien en los tres) para tener un entorno ágil y rápido:

- Quarkus: “./mvnw compile quarkus:dev”
- Micronaut: “./mvnw mn:run”
- SpringBoot: el modo desarrollo requerirá incluirá la dependencia DevTools y además arrancar la aplicación desde un IDE con plugin SpringBoot (Eclipse, IntelliJ, VS Code..).

Además, todos ellos cuentan con una extensión/dependencia para realizar una integración fácil con kubernetes:

- Quarkus: quarkus-kubernetes
- Micronaut: io.micronaut.kubernetes
- SpringBoot: Spring Cloud Kubernetes

Objetivos

El principal objetivo es cubrir mediante diferentes análisis cómo implementar los principales patrones de diseño de Kubernetes a la hora de desarrollar aplicaciones cloud-native en Java. Desde cómo implementar un operador de Kubernetes en Java, realizar una monitorización apropiada u optimizar el uso de la JVM en contenedores docker, hasta implementar en detalle patrones como health-check, lectura de configmaps o exposición de métricas de la JVM y customizadas en los diferentes frameworks mencionados con la finalidad de realizar una comparativa entre los mismos.

La idea no es tanto determinar si un framework es mejor que otro, ya que los tres son realmente muy parecidos, si no detallar el cómo se puede hacer y realizar una comparativa.

Análisis realizado

El trabajo llevado a cabo se divide en cinco secciones diferentes. Pese a que cada análisis en profundidad se encuentra en los diferentes directorios del repositorio GIT de la Universidad (https://github.com/MasterCloudApps-Projects/Java-Kubernetes), se va a detallar a continuación en qué consiste cada uno y qué pasos se han llevado a cabo.

Análisis health checks en diferentes frameworks java

Este primer análisis se centra en averiguar qué nos ofrecen actualmente los diferentes frameworks de java orientados a microservicios y cloud para facilitar la implementación de health checks en kubernetes.

Los requisitos para poder no solo entenderlo en profundidad, si no para también seguir los pasos indicados a lo largo de la documentación con la finalidad de replicar el resultado en un clúster de Kubernetes, son:

- Conocimientos básicos de kubernetes
- Maven
- Cluster local de kubernetes, por ejemplo Minikube
- Kubectl

Aunque los conceptos principales de kubernetes se darán por conocidos, ya que el objetivo de este proyecto no es explicar cómo funciona, ya que para eso ya hay miles de recursos en internet, aquí va una breve introducción:

> *Uno de los conceptos básicos de kubernetes son los health checks. La idea es muy básica, ya que kubernetes sabrá cuándo un pod se encuentra sano basándose en estos "checks". Por un lado, el "Liveness probe" (el pod sigue vivo? debería reiniciarse?), y por otro lado, el "Readiness probe" (está el pod en condiciones para recibir tráfico?). Mediante archivos de configuración yaml, se indicará a kubernetes dónde se encuentran los endpoints a los que debe llamar dentro de la aplicación para realizar los mencionados chequeos.*

Los frameworks java en los que se va a centrar el análisis son Quarkus, Micronaut y Springboot, orientados a cloud y microservicios, y ver qué ofrecen para facilitar la vida a la hora de implementar health checks en una aplicación.

Una vez se descargan los proyectos de las correspondientes webs, y teniendo maven instalado, ya serían proyectos funcionales con las dependencias seleccionadas que se podrían ejecutar. El análisis será idéntico para cada uno de los frameworks:

1. Generar el proyecto boilerplate desde la web, añadiendo las dependencias necesarias tanto para tener disponibles los health checks como un datasource para la conexión a una base de datos postgres. La idea es que el simple hecho de tener un datasource activo, ya debería de impactar en los health checks de los diferentes frameworks sin tener que configurar nada adicionalmente, ya que, si la conexión a base de datos no funciona, el pod no debería de recibir tráfico alguno desde kubernetes.

2. Añadir yamls básicos tanto de “deployment” como de “service” para el despliegue en el clúster de kubernetes local (usaré minikube para probar los ejemplos).
3. Desplegar el servicio y analizar la respuesta del endpoint “/health” que incluye la configuración “out of the box”, simplemente añadiendo el datasource de postgres y desplegando en minikube. En principio todos los boilerplate vienen con un Dockerfile o similar incluido.
4. Por último, implementar la interfaz de los health checks de cada framework con el fin de poder crear uno totalmente customizado, simulando la comprobación de un servicio customizado que pudiésemos tener en nuestro negocio. Se desplegará en minikube el código con el nuevo check y se analizará cómo impacta en el endpoint “/health”.

Como comparativa final, se deja una tabla en la que se puede observar en qué endpoints exponen los health-checks cada framework por defecto, y qué tipo de respuestas JSON ofrece cada uno:

Quarkus	Micronaut	Spring Boot
/health/live	/info	/actuator/health/liveness
/health/ready	/health	/actuator/health/readiness

```
{
    "status": "UP",
    "checks": [
        {
            "name": "Custom-service",
            "status": "UP"
        },
        {
            "name": "Database connections health check",
            "status": "UP"
        }
    ]
}
```

```
{
   "name":"healthcheck",
   "status":"UP",
   "details":{
      "Custom-service":{
         "name":"healthcheck",
         "status":"UP"
      },
      "jdbc":{
         "name":"healthcheck",
         "status":"UP",
         "details":{
            "jdbc:postgresql://postgresdb:5432/postgres":{
               "name":"healthcheck",
               "status":"UP",
               "details":{
                  "database":"PostgreSQL",
                  "version":"12.3 (Debian 12.3-1.pgdg100+1)"
               }
            }
         }
      }
   }
}
```

```
{
   "status":"UP",
   "components":{
      "customHealthCheck":{
         "status":"UP",
         "details":{
            "Custom-service":"Available"
         }
      },
      "db":{
         "status":"UP",
         "details":{
            "database":"PostgreSQL",
            "validationQuery":"isValid()"
         }
      },
      "livenessState":{
         "status":"UP"
      },
      "ping":{
         "status":"UP"
      },
      "readinessState":{
         "status":"UP"
      }
   },
   "groups":[
      "liveness",
      "readiness"
   ]
}
```

5. Tabla comparativa health-checks por framework

Todo el código y el análisis en profundidad se encuentra disponible en el directorio health-checks del repositorio mencionado anteriormente (https://github.com/MasterCloudApps-Projects/Java-Kubernetes/tree/master/health-checks).

Análisis uso de configmaps en diferentes frameworks java

Este segundo análisis se centra en averiguar qué nos ofrecen actualmente los principales frameworks de java orientados a microservicios y cloud a la hora de interactuar con configmaps en kubernetes, evaluando la posibilidad de detectar cambios "en caliente".

En todos los frameworks, siempre se tendrá un archivo application.properties o application.yaml, donde se guardará la configuración de la aplicación (desde los parámetros de configuración a un datasource, a la frecuencia con la que se ejecuta un timer). A la hora de desplegar la aplicación en un entorno en concreto, las variables de entorno sobrescribirán dichas propiedades; es decir, si en el archivo application.properties existe una propiedad properties.fubar.foo, y al desplegar la aplicación, existe una variable de entorno PROPERTIES_FUBAR_FOO configurada, ésta sobrescribirá el valor de la variable en el archivo properties. Esto es muy útil, ya que se podrán configurar los diferentes parámetros según el entorno en el que se despliegue la aplicación.

Aquí es donde entran en juego los configmaps. Mediante archivos de configuración yml, se podrá crear y asignar en kubernetes un set de datos para uno o varios deployments, actuando como variables de entorno para los mismos (y por tanto sobreescribiendo los valores de las properties en caso de haberlas), haciendo que configurar los contenedores orquestados bajo kubernetes se facilite de muchas maneras.

Los requisitos para poder no solo entenderlo en profundidad, si no para también seguir los pasos indicados a lo largo de la documentación con la finalidad de replicar el resultado en un clúster de Kubernetes, son:

- Conocimientos básicos de kubernetes
- Maven
- Cluster local de kubernetes, por ejemplo Minikube
- Kubectl

El análisis será idéntico para cada uno de los frameworks:

1. Generar el proyecto boilerplate desde la web, añadiendo las dependencias necesarias para comunicarse con la API de kubernetes, así como servir un endpoint http. En primera instancia se devolverá por http el valor de dos propiedades configuradas en el archivo application.properties.
2. Crear yamls básicos tanto para añadir el configmap (con valores diferentes que deberán sobrescribir las propiedades) en kubernetes como para asignar los permisos necesarios al deployment para el acceso a la API de kubernetes. Despliegue en un clúster kubernetes local (se utilizará minikube para probar los ejemplos). Tanto el deployment como el service de cada aplicación se darán ya de base, puesto que no son relevantes en este análisis.
3. Comprobar si la respuesta del endpoint devuelve los valores de las variables del configmap, y no las originales del archivo properties. Adicionalmente, y si el framework lo permite, hacer un cambio en el configmap de forma manual en kubernetes y comprobar cómo la aplicación detecta el cambio y reemplaza los valores “en caliente”.

4. Por último, explorar qué otras posibilidades ofrece cada framework en relación a la dependencia de kubernetes.

Todo el código y el análisis en profundidad se encuentra disponible en el directorio health-checks del repositorio mencionado anteriormente (https://github.com/MasterCloudApps-Projects/Java-Kubernetes/tree/master/reload-configmap).

Optimización de recursos de la JVM en Kubernetes

Uno de los principales retos al desplegar servicios en kubernetes de manera satisfactoria es ajustar apropiadamente los recursos. Concretamente, las aplicaciones java necesitan especial atención ya que la JVM se puede volver insaciable en algunos casos, sufriendo muy frecuentemente tanto OOM kills como tiempos de arranque muy lentos.

Este tercer análisis se centra en el consumo de memoria y CPU de un microservicio springboot básico (con health check y una conexión a postgres, todo funcionando en kubernetes), tanto en la fase de arranque como de ejecución de la aplicación, con la intención de crear y ajustar ambos parámetros con valores apropiados tanto para resources.requests como para resources.limits.

Los requisitos para poder no solo entenderlo en profundidad, si no para también seguir los pasos indicados a lo largo de la documentación con la finalidad de replicar el resultado en un clúster de Kubernetes, son:

- Conocimientos básicos de kubernetes
- Conocimientos básicos de la JVM
- Maven
- Cluster local de kubernetes, por ejemplo Minikube
- Kubectl

Aunque se dan por conocidos estos conceptos de kubernetes, un repaso a qué representan estos valores hará que el análisis se entienda un poco mejor.

> *Al pedir a Kubernetes que ejecute una aplicación, el scheduler de kubernetes buscará un nodo en el cluster donde los pods de la aplicación puedan ser ejecutados. El principal factor para determinar si el pod puede ser ejecutado en un nodo será si tiene suficiente memoria y CPU; y aquí es donde la sección resources del yaml de kubernetes entra en juego, con dos apartados, requests y limits.*

> *Requests representará los valores mínimos que cada pod necesitará para arrancar de manera satisfactoria. En caso de no encontrar un nodo que cumpla dichos requisitos, el pod quedará marcado como unschedulable.*

> *Una vez el pod se encuentre en ejecución, podrá requerir memoria y CPU adicional al nodo; aquí es donde entra en juego el parámetro limits, estableciendo el techo máximo tanto de memoria como de CPU para cada pod.*

La motivación por tanto es clara, proporcionando unos valores adecuados de resources en el deployment de Kubernetes, se asegurará la existencia de recursos suficientes en los nodos donde se ejecutan los pods de la aplicación, es decir, se proporcionará al cluster de Kubernetes la oportunidad de realizar una eficiente gestión de los recursos CPU y memoria.

Se parte de un servicio springboot, con un datasource para postgres y un endpoint, haciendo uso de actuator, para health checks. El yaml de configuración de la aplicación en Kubernetes no contiene por

el momento la sección resources. Tanto Postgres como la aplicación se ejecutarán en un clúster de Kubernetes local usando Minikube. A partir de este punto, se realizarán los siguientes pasos:

1. Establecer los recursos del servicio haciendo uso del addon de Minikube “metrics-server”, añadiéndolos al yaml inicial de la aplicación Springboot.
2. Optimizar el uso de memoria de la JVM en contenedores docker.
3. Troubleshooting de pods java en kubernetes.

Todo el código y el análisis en profundidad se encuentra disponible en el directorio health-checks del repositorio mencionado anteriormente (https://github.com/MasterCloudApps-Projects/Java-Kubernetes/tree/master/resources).

Diseño de un operador de Kubernetes en Java

En Kubernetes, el concepto de operador es similar al de un controlador, el cuál observa en bucle el estado del clúster y realiza cambios cuando es necesario (debe llevar el estado actual al estado deseado). El más conocido es probablemente el deployment controller; cada vez que se realiza un nuevo despliegue, se puede observar cómo este crea un pod deploy que se encarga de crear tantas réplicas de la aplicación como la especificación indique. Estos controladores se ejecutan en el control plane de Kubernetes, pero uno customizado como el que se quiere diseñar puede ser ejecutado en cualquier sitio.

Por tanto, un operador será un controlador, pero especializado con conocimiento específico de negocio, interactuando con el clúster de kubernetes para crear, configurar y gestionar instancias de la aplicación, un recurso específico. Esta manera de interactuar del operador con el clúster de kubernetes será a través de la Kubernetes API Server, el frontend del clúster a través del cual se podrán validar y configurar objetos como pods, services y replicationcontrollers entre otros.

Por ejemplo, la conocida kubectl es una herramienta de línea de comando para interactuar exactamente con esta API. Solo que en este caso se quiere interactuar con la API vía HTTP y desde un pod del propio kubernetes.

Los requisitos para poder no solo entenderlo en profundidad, si no para también seguir los pasos indicados a lo largo de la documentación con la finalidad de replicar el resultado en un clúster de Kubernetes, son:

- Conocimientos básicos de kubernetes
- Maven
- Cluster local de kubernetes, por ejemplo Minikube
- Kubectl

Se realizarán los siguientes pasos:

1. Análisis del diseño de un operador Java, tanto para la autorización de la lectura de la API de Kubernetes desde la aplicación Java, como los posibles clientes Java ya existentes (io.kubernetes vs io.fabric8) y qué interfaces deberían de implementar cada uno.
2. Creación y despliegue del operador Java en el clúster de Kubernetes, haciendo un repaso de un repositorio muy interesante de Nicolas Frankel.

Todo el código y el análisis en profundidad se encuentra disponible en el directorio health-checks del repositorio mencionado anteriormente (https://github.com/MasterCloudApps-Projects/Java-Kubernetes/tree/master/kubernetes-client).

Monitorización y visualización de logs de una aplicación java en kubernetes

Este quinto y último análisis se centra en averiguar qué nos ofrecen actualmente los diferentes frameworks de java orientados a microservicios y cloud para facilitar la exposición de métricas a un operador prometheus tanto de la JVM como customizadas.

Los requisitos para poder no solo entenderlo en profundidad, si no para también seguir los pasos indicados a lo largo de la documentación con la finalidad de replicar el resultado en un clúster de Kubernetes, son:

- Conocimientos básicos de kubernetes
- Maven
- Cluster local de kubernetes, por ejemplo Minikube
- Kubectl

De nuevo, los frameworks java en los que se va a centrar el análisis son Quarkus, Micronaut y Springboot.

Hoy en día la monitorización de una aplicación es algo esencial para cualquier equipo. Además, la visualización de logs es crucial en el análisis de errores e incidencias, y sobre todo cuando existen diversos entornos, la centralización de todos estos datos es indispensable.

Con un uso muy extendido, open source e inicialmente desarrollado por SoundCloud en 2012 y más tarde adoptado por la CNCF (Cloud Native Computing Foundation), Prometheus se postula como una de las principales opciones a la hora de llevar a cabo estas tareas de lectura de métricas para monitoreo y sistema de alertas. Además, se complementa muy bien con Grafana, ya que se puede establecer Prometheus como datasource y crear dashboards gráficos muy fácilmente.

Para lectura de logs, también open source y desarrollado por Grafana, Loki se autodefine como “like Prometheus, but for logs”. No hace falta mencionar que se integra perfectamente con Grafana.

Una vez se prepare todo el stack de monitorización y visualización de logs y, por tanto, Prometheus, Grafana y Loki estén funcionando en el clúster de Kubernetes (se utilizarán Helm charts para la instalación), el análisis será idéntico para cada uno de los frameworks:

1. Generar el proyecto boilerplate desde la web, añadiendo las dependencias necesarias para exponer las métricas de la JVM utilizando micrometer o similar.
2. Crear una clase ScrapingMetrics, simulando el comportamiento que tendría el scraping de una web externa, con el objetivo de exponer dos métricas customizadas, una de tipo Counter (cuántas veces se ha realizado el scraping desde el inicio de la app) y otra de tipo Gauge (tiempo que ha tardado en realizarse el último scraping).
3. Añadir yamls básicos tanto de deployment como de service para el despliegue en el clúster de kubernetes local (usaré minikube para probar los ejemplos). Crear a su vez un yaml adicional de tipo ServiceMonitor, donde se le indicará al operador de prometheus dónde y cómo leer las métricas de la aplicación. Desplegar el servicio.
4. Configurar un dashboard en Grafana mediante el el datasource de Prometheus mostrando las métricas tanto de la JVM como customizadas. Establecer una alerta para una de las métricas.

5. Por último, añadir un segundo panel al dashboard de Grafana mediante el datasource de Loki mostrando los logs de la aplicación junto a las métricas.

Todo el código y el análisis en profundidad se encuentra disponible en el directorio health-checks del repositorio mencionado anteriormente (https://github.com/MasterCloudApps-Projects/Java-Kubernetes/tree/master/metrics).

Conclusiones

En primer lugar, la principal conclusión es que los tres frameworks, tanto Quarkus, como Micronaut como SpringBoot han permitido implementar los principales patrones de diseño Kubernetes sin mayor problema o dificultad. Todos ofrecen con garantías lo necesario para desarrollar de una manera cómoda. No obstante, seguiría escogiendo SpringBoot como solución standard a la hora de desarrollar microservicios Java; cuenta con la comunidad más completa, de lejos la mayor madurez de la plataforma y muchísimos ejemplos online para troubleshooting por parte del equipo de desarrollo. Tanto Quarkus como Micronaut se encuentran bajo un desarrollo muy fuerte y creo que romperán APIs con mucha facilidad todavía.

Dicho esto, a la hora de programar, Quarkus ofrece una mayor productividad frente al resto. Todo es muy fácil y sencillo, es rapidísimo y se hace mucho con muy poco código. Además, implementa MicroProfile, lo cual puede ser una alternativa muy buena si vienes de JavaEE y necesitas ciertas compatibilidades.

Si, además, se busca hacer uso de GraalVM, Quarkus sería de lejos la mejor opción, ya que "ha nacido" pensando en eso. Todo es muy sencillo y tiene un rendimiento buenísimo.

Como inconveniente, en el análisis de consumo de configmaps desde aplicaciones Java, Quarkus ha sido el único incapaz de conseguir "hot-swap" de variables de entorno mediante detección de cambios en el configmap durante tiempo de ejecución.

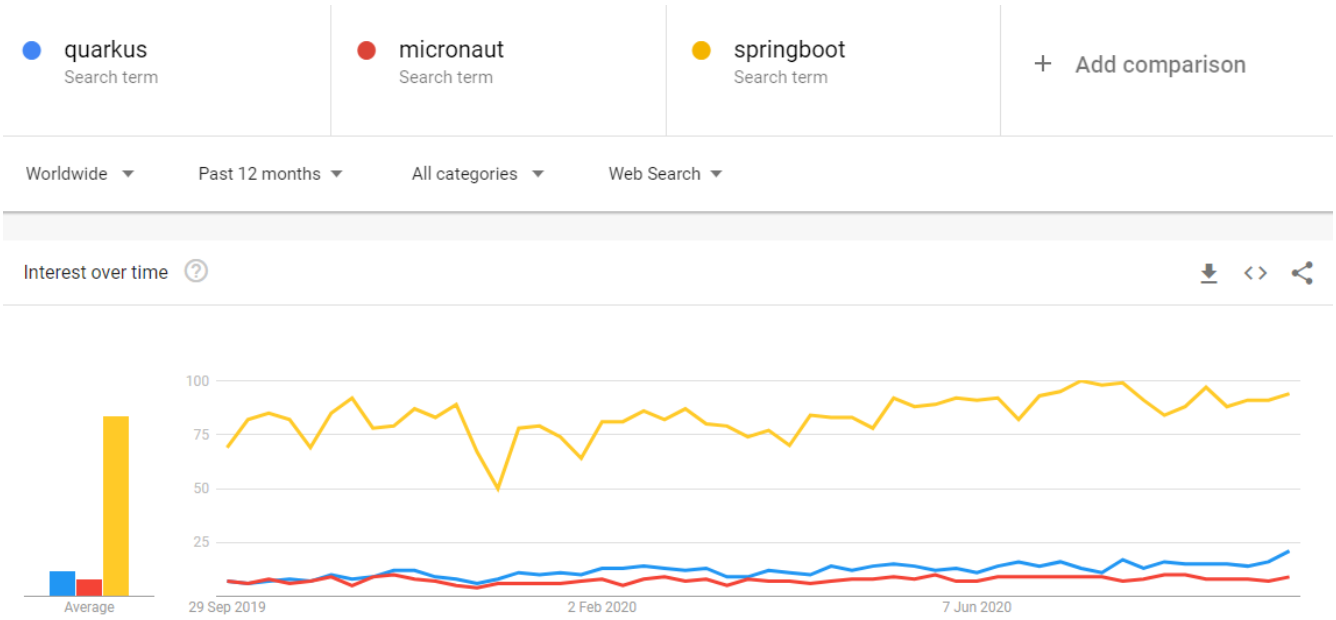
Micronaut, pese a tener la menor comunidad y menos ejemplos de código en su documentación, desde que han liberado su versión 2.0 se han puesto a la par en cuanto a rendimiento y posibilidades. Sin embargo, aunque el uso de memoria por parte de Quarkus sea el mejor, seguido muy de cerca por Micronaut (SpringBoot el peor), trabajando bajo carga Micronaut parece ser el que más desperdicia recursos, por lo que pasaría al tercer puesto en un pico de trabajo, manteniéndose Quarkus al frente, tanto trabajando bajo la JVM como GraalVM.

A la hora de escoger un cliente Kubernetes en Java, Fabric8 es de lejos la mejor solución, ya que tiene mucho mayor soporte por parte de la comunidad que el resto, es el más sencillo y es muy completo.

En lo referente a exposición de métricas, ha sido realmente sencillo en cualquiera de los tres frameworks, tanto de la JVM como customizadas. Además, el único que no implementa Micrometer (Quarkus) ofrece métricas compatibles con dashboards hechos para Micrometer, lo que demuestra que Quarkus quiere ganar mercado ofreciendo migraciones muy sencillas tanto si vienes de Spring como si vienes de JavaEE. Cabe destacar que Micronaut es el más completo a la hora de exponer métricas a los diferentes operadores, aunque todos soportan Prometheus de base.

Se nota que los tres frameworks se siguen la pista de cerca, ya que todos intentan sacar lo que tienen el resto para no quedarse atrás. Incluso se pueden observar issues de Github en cualquiera de los tres donde se mencionan a los otros frameworks como referencia a la hora de implementar algo. Esta carrera por ser el framework de referencia "para cloud".

No obstante, y pese al gran crecimiento que está experimentando Quarkus en este último año, SpringBoot es todavía de lejos el más usado y Micronaut se sitúa a la cola.



6. Tabla comparativa Quarkus vs SpringBoot vs Micronaut

Bibliografía

- [1] JVM in a Container-Merikan Blog. Retrieved from https://merikan.com/2019/04/jvm-in-a-container/
- [2] Kubernetes Demystified: Restrictions on Java Application Resources - Alibaba Cloud Community. Retrieved from https://www.alibabacloud.com/blog/kubernetes-demystified-restrictions-on-java-application-resources_594108
- [3] Sizing Kubernetes pods for JVM apps without fearing the OOM Killer. Retrieved from https://srvaroa.github.io/jvm/kubernetes/memory/docker/oomkiller/2019/05/29/k8s-and-java.html
- [4] Java Application Optimization on Kubernetes on the Example of a Spring Boot Microservice | by Stephan Hartmann | FAUN | Medium. Retrieved from https://medium.com/faun/java-application-optimization-on-kubernetes-on-the-example-of-a-spring-boot-microservice-cf3737a2219c
- [5] Playing with the JVM inside Docker Containers | Nebrass's Homepage. Retrieved from https://blog.nebrass.fr/playing-with-the-jvm-inside-docker-containers/
- [6] Kubernetes ConfigMap Configuration and Reload Strategy | by Eresh Gorantla | The Startup | Medium. Retrieved from https://medium.com/swlh/kubernetes-configmap-confuguration-and-reload-strategy-9f8a286f3a44
- [7] Spring Cloud Kubernetes. Retrieved from https://cloud.spring.io/spring-cloud-static/spring-cloud-kubernetes/2.1.0.RC1/single/spring-cloud-kubernetes.html
- [8] Quarkus - Kubernetes extension. Retrieved from https://quarkus.io/guides/kubernetes
- [9] Micronaut Kubernetes. Retrieved from https://micronaut-projects.github.io/micronaut-kubernetes/2.0.0/guide/index.html
- [10] Spring Cloud Kubernetes. Retrieved from https://spring.io/projects/spring-cloud-kubernetes
- [11] Spring Cloud Kubernetes. Retrieved from https://cloud.spring.io/spring-cloud-kubernetes/reference/html/#configuration-properties
- [12] Slides – Create and share presentations online. Retrieved from https://slides.com/
- [13] Monitoring a Spring Boot app in Kubernetes - What I learned from Devoxx Belgium 2019 - LINE ENGINEERING. Retrieved from https://engineering.linecorp.com/en/blog/monitoring-a-spring-boot-app-in-kubernetes-what-i-learned-from-devoxx-belgium-2019/
- [14] Trying Prometheus Operator with Helm + Minikube | by Eduardo Baitello | FAUN | Medium. Retrieved from https://medium.com/faun/trying-prometheus-operator-with-helm-minikube-b617a2dccfa3
- [15] The Prometheus Operator: Managed Prometheus setups for Kubernetes | CoreOS. Retrieved from https://coreos.com/blog/the-prometheus-operator.html
- [16] prometheus-operator 0.38.1 for Kubernetes | Helm Hub | Monocular. Retrieved from https://hub.helm.sh/charts/stable/prometheus-operator
- [17] Monitoring Spring Boot Application with Prometheus and Grafana on Kubernetes | Niraj Sonawane. Retrieved from https://nirajsonawane.github.io/2020/05/17/Monitoring-Spring-Boot-Application-with-Prometheus-and-Grafana-on-Kubernetes/
- [18] Spring Metrics. Retrieved from https://docs.spring.io/spring-metrics/docs/current/public/prometheus
- [19] Micrometer Application Monitoring. Retrieved from https://micrometer.io/docs/concepts
- [20] Micrometer: Spring Boot 2's new application metrics collector. Retrieved from https://spring.io/blog/2018/03/16/micrometer-spring-boot-2-s-new-application-metrics-collector

- [21] Spring Initializr. Retrieved from https://start.spring.io/
- [22] Kubernetes Monitoring: Monitor your apps in Kubernetes with Prometheus and Spring Boot – IBM Developer. Retrieved from https://developer.ibm.com/technologies/containers/tutorials/monitoring-kubernetes-prometheus/
- [23] Monitor Spring Boot Apps using Prometheus on Kubernetes | by Alex Cheng | VividCode | Medium. Retrieved from https://medium.com/vividcode/monitor-spring-boot-apps-using-prometheus-on-kubernetes-a7c1b58c8511
- [24] Prometheus Operator — How to monitor an external service | by Ido Braunstain | DevOps College. Retrieved from https://devops.college/prometheus-operator-how-to-monitor-an-external-service-3cb6ac8d5acb
- [25] Kube-Proxy endpoint connection refused · Issue #16476 · helm/charts. Retrieved from https://github.com/helm/charts/issues/16476
- [26] Helm | Grafana Labs. Retrieved from https://grafana.com/docs/loki/latest/installation/helm/
- [27] grafana/loki: Like Prometheus, but for logs.. Retrieved from https://github.com/grafana/loki
- [28] Loki Setup for Kubernetes cluster logging | by Prakash Singh | Medium. Retrieved from https://medium.com/@psingh.singh361/loki-setup-for-kubernetes-cluster-b7403e137591
- [29] Loki Helm Chart | loki. Retrieved from https://grafana.github.io/loki/charts/
- [30] CNCF Prometheus Project Journey Report | Cloud Native Computing Foundation. Retrieved from https://www.cncf.io/cncf-prometheus-project-journey/
- [31] Quarkus - MicroProfile Metrics. Retrieved from https://quarkus.io/guides/microprofile-metrics
- [32] Micronaut Micrometer. Retrieved from https://micronaut-projects.github.io/micronaut-micrometer/3.0.0/guide/index.html
- [33] ¿Qué es Docker?. Retrieved from https://www.redhat.com/es/topics/containers/what-is-docker
- [34] Docker (software) - Wikipedia, la enciclopedia libre. Retrieved from https://es.wikipedia.org/wiki/Docker_(software)
- [35] Utilizar Docker: beneficios, estadísticas y historias de éxito | Apiumhub. Retrieved from https://apiumhub.com/es/tech-blog-barcelona/beneficios-de-utilizar-docker/
- [36] Kubernetes - Wikipedia, la enciclopedia libre. Retrieved from https://es.wikipedia.org/wiki/Kubernetes
- [37] What is Kubernetes? | Kubernetes. Retrieved from https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
- [38] CNCF Survey: Use of cloud native technologies in production has grown over 200% | Cloud Native Computing Foundation. Retrieved from https://www.cncf.io/blog/2018/08/29/cncf-survey-use-of-cloud-native-technologies-in-production-has-grown-over-200-percent/
- [39] China vs. the World: A Kubernetes and Container Perspective – The New Stack. Retrieved from https://thenewstack.io/china-vs-the-world-a-kubernetes-and-container-perspective/
- [40] What is Kubernetes?. Retrieved from https://www.redhat.com/en/topics/containers/what-is-kubernetes?exa47498546=&adobe_mc_sdid=SDID%3D06BE018C930217D0-46F63992423030E0%7CMCORGID%3D945D02BE532957400A490D4C%40AdobeOrg%7CTS%3D1601150119&adobe_mc_ref=https%3A%2F%2Fwww.google.com%2F
- [41] Which Java Microservice Framework Should You Choose in 2020? | by Matthias Graf | Better Programming | Medium. Retrieved from https://medium.com/better-programming/which-java-microservice-framework-should-you-choose-in-2020-4e306a478e58

- [42] quarkus, micronaut, springboot - Explore - Google Trends. Retrieved from https://trends.google.com/trends/explore?q=quarkus,micronaut,springboot