UNIVERSIDAD POLITÉCNICA SALESIANA

SEDE CUENCA

CARRERA DE INGENIERÍA DE SISTEMAS

Trabajo de titulación previo
a la obtención del título de
Ingeniero de Sistemas

PROYECTO TÉCNICO:

“DESARROLLO E IMPLEMENTACIÓN DE UNA ARQUITECTURA DEVOPS PARA UN SISTEMA WEB BASADO EN MICROSERVICIOS EN INFRAESTRUCTURAS BASADAS EN CÓDIGO”

AUTOR:

BRAULIO FERNANDO CUSCO MEJÍA

TUTOR:

ING. DIEGO FERNANDO QUISI PERALTA

CUENCA - ECUADOR

2022

CESIÓN DE DERECHOS DE AUTOR

Yo, Braulio Fernando Cusco Mejía con documento de identificación Nº 0105340541, manifiesto mi voluntad y cedo a la Universidad Politécnica Salesiana la titularidad sobre los derechos patrimoniales en virtud de que soy autor del trabajo de titulación: **"DESARROLLO E IMPLEMENTACIÓN DE UNA ARQUITECTURA DEVOPS PARA UN SISTEMA WEB BASADO EN MICROSERVICIOS EN INFRAESTRUCTURAS BASADAS EN CÓDIGO"**, mismo que ha sido desarrollado para optar por el título de: *Ingeniero de Sistemas*, en la Universidad Politécnica Salesiana, quedando la Universidad facultada para ejercer plenamente los derechos cedidos anteriormente.

En aplicación a lo determinado en la Ley de Propiedad Intelectual, en mi condición de autor me reservo los derechos morales de la obra antes citada. En concordancia, suscribo este documento en el momento que hago entrega del trabajo final en formato digital a la Biblioteca de la Universidad Politécnica Salesiana.

Cuenca, enero de 2022.

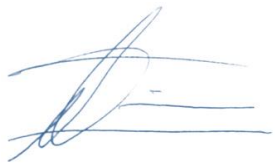
Braulio Fernando Cusco Mejía

C.I. 0105340541

CERTIFICACIÓN

Yo, declaro que bajo mi tutoría fue desarrollado el trabajo de titulación: **"DESARROLLO E IMPLEMENTACIÓN DE UNA ARQUITECTURA DEVOPS PARA UN SISTEMA WEB BASADO EN MICROSERVICIOS EN INFRAESTRUCTURAS BASADAS EN CÓDIGO"**, realizado por Braulio Fernando Cusco Mejía, obteniendo el *Proyecto Técnico* que cumple con todos los requerimientos estipulados por la Universidad Politécnica Salesiana.

Cuenca, enero de 2022.



Diego Fernando Quisi Peralta

C.I. 0104616461

DECLARATORIA DE RESPONSABILIDAD

Yo, Braulio Fernando Cusco Mejía con documento de identificación Nº 0105340541, autor del trabajo de titulación: **"DESARROLLO E IMPLEMENTACIÓN DE UNA ARQUITECTURA DEVOPS PARA UN SISTEMA WEB BASADO EN MICROSERVICIOS EN INFRAESTRUCTURAS BASADAS EN CÓDIGO"**, certifico que el total contenido del *Proyecto Técnico*, es de mi exclusiva responsabilidad y autoría.

Cuenca, enero de 2022.

Braulio Fernando Cusco Mejía

C.I. 0105340541

AGRADECIMIENTO

En primera instancia agradezco a Dios por todas las oportunidades que se me han presentado a lo largo de la carrera, de igual manera agradezco a la Universidad Politécnica Salesiana y a cada uno de los docentes con lo que compartí el aula donde pusieron todo su esfuerzo y talento para compartir sus conocimientos, apoyo para ser un profesional, también un agradecimiento especial para el Ing. Diego Quisi, quien me apoyo durante todo el desarrollo del proyecto.

DEDICATORIA

Dedico de manera especial a mi madre **NARCISA**, a mi padre **JUAN**, y mi hermana **DAYANNA** quienes fueron la base principal para completar esta meta, por su apoyo y todo su esfuerzo realizado durante todo este tiempo, sin sus palabras de aliento y motivación nada de esto hubiese sido posible.

A mi familia y amigos que siempre me ayudaron a seguir adelante con sus sanos consejos.

ÍNDICE DE CONTENIDO

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

RESUMEN

Se muestra una de las varias maneras que existen para automatizar la construcción y despliegue de aplicaciones en entornos de desarrollo o producción, usando la metodología de desarrollo de Software DevOps, actualmente en Ecuador es poco conocida esta metodología, muchas empresas aún trabajan de manera tradicional. En consecuencia, esto se trata de solucionar con DevOps minimizando costos de entrega, tiempos de desarrollo y brindar un valor agregado a los usuarios, para ello se definió una arquitectura que implementa una herramienta OpenSource que están apoyadas por grandes empresas como, por ejemplo, Google, Gitlab y por comunidades oficiales, aportando un gran valor a todas estas herramientas.

La arquitectura está formada por Kubernetes, que administrará a gran escala las aplicaciones que serán desplegadas las imágenes que previamente han pasado las Pruebas, el motor principal de contendores es Docker. La arquitectura general consta de varias partes empezando por un balanceador de carga que será el único acceso que responderá peticiones al servidor o al cliente, un objeto Kubernetes de tipo Ingress el cual se encarga de reenviar el tráfico al servicio correspondiente, además servicios de tipo ClusterIp que responderán al Ingress cada vez que exista una nueva solicitud y que estarán relacionados a cada Deployment uno para la base de datos, otro para el servidor y para el cliente, para persistir la información de la base de datos se agregó un Volumen .Por otro lado, para la integración continua y el despliegue continúo se utiliza la herramienta Gitlab CI, que principalmente se definen los pasos para automatizar la compilación, construcción y despliegue de la aplicación.

Finalmente, para lo que son pruebas se desarrolló dos tipos de pruebas, una prueba de carga y de Stress, los resultados generales son buenos en cuestiones de rendimiento puesto que el Clúster en el que todo se desplego es de características de Hardware no recomendable para producción más que nada es para desarrollo. Para la prueba de carga se hace uso de Jmeter una herramienta que es pionera en cuestiones rendimiento y pruebas, los resultados obtenidos en base a varias ejecuciones de la misma prueba muestran porcentajes similares para tiempos de respuesta el 0.01% de las peticiones terminaron con algún error hasta un máximo de 0.08%, es decir del total de peticiones 8 de las 10000 no se ejecutaron y 9992 se ejecutaron. Para lo que es la prueba de Stress simplemente se realiza de manera infinita peticiones a la aplicación del cliente, desde un contendor se ejecuta la prueba, además de eso se escala el número de contenedores a 30, 40 y 50, lo que hace que esta prueba no solamente se ponga a prueba el Pod de la aplicación del cliente sino al Nodo Worker donde se ejecutan los Pods, todas las métricas de consumo se ven reflejadas en la aplicación Grafana que muestra mediante gráficos las métricas del Clúster, los resultados generales son muy buenos para el Hardware que se tiene, no existieron problemas de caídas de nodos, ni mucho menos de contenedores.

CAPÍTULO 1 INTRODUCCIÓN AL ESTADO DEL ARTE

1.1 INTRODUCCIÓN

Cuando se trata de desarrollo y despliegue de aplicaciones tanto del lado del servidor como del lado de cliente es inevitable omitir tener un servidor para cada uno y en ocasiones se tienen más servidores para realizar las pruebas antes del despliegue definitivo, esto conlleva problemas, entre ellos lo económico, aumentando los costos para la empresa, dificultad de mantener varios servidores a la vez, problemas en su infraestructura, en caso de problemas de software su difícil solución y pérdida de tiempo. Para solucionar o al menos disminuir la cantidad de todos estos problemas las empresas grandes están optando por migrar o reconstruir sus arquitecturas a partir de Kubernetes, Docker principalmente en IaaS (Infraestructura como servicio).

Actualmente las empresas han optado por la virtualización de todos sus servicios a los clientes, pero existe el problema de rendimiento, claro esto es fácil de solucionar, se compra o alquila más recursos, que genera más gasto de dinero. Por lo contrario, la caída de sus servicios en virtualización no es tan frecuente con una buena infraestructura.

En caso de la caída del servicio existe una réplica como primario y secundario o en el mejor de los casos se levantaría una nueva instancia que funcionara como primaria, en este caso el secundario entraría en función hasta que los administradores de red investiguen el problema y lo solucionen, mientras tanto el rendimiento del servicio es menor y puede llevar a pérdidas económicas, ahora por el lado de Kubernetes y Docker, existen las réplicas que se configuran en los manifiestos que son archivos de configuración. Además, se puede ejecutar directamente en línea de comandos para hacer un Deployment de una imagen preconfigurada con los requisitos necesarios, en el manifiesto se indica el número de réplicas, por ejemplo 3, en caso de la caída de una, deja de funcionar o

simplemente se elimina, Kubernetes se da cuenta y levanta una nueva instancia sin necesidad de disminuir el rendimiento, de esta manera siempre mantiene el servicio activo.

1.2 CASOS DE ÉXITO

1.2.1 CASO DE ÉXITO BARCLAYS

Analizando un caso de éxito sobre la implementación de DevOps, se tiene un claro ejemplo, Barclays, la compañía de servicios financieros que opera a nivel mundial tomó la gran decisión de adoptar esta cultura de DevOps. “Necesitamos comercializar los productos más rápido que nunca y contar con mayor capacidad de respuesta a las tendencias del mercado” (Cashmore, 2017). Esta entidad cambió todo su entorno de TI, para lograr aprovechar al máximo las herramientas que ellos tenían, “Deseamos ser más dinámicos en la forma de producir aplicaciones y hacer un mejor uso de nuestro hardware y software subyacentes, además de nuestro personal” (Cashmore, 2017). Por ello Barclays hizo DevOps para que sus equipos trabajen de manera más organizada, tanto el equipo técnico y el equipo comercial, de esta manera ellos puedan cumplir las expectativas que tenían sus clientes a través del desarrollo continuo, Barclays cambió su middleware tradicional por Red Hat OpenShift Container Platform como una estrategia de nube, con ayuda de esta tecnología Barclays trabaja de una manera más ágil en la que pueden ofrecer a sus clientes y trabajadores nuevas actualizaciones de manera más rápida; en consecuencia, han logrado disminuir los tiempos de entrega, la implementación de código llevaba un proceso de 56 días, ahora les toma 4 semanas en implementarlo, de esta forma Barclays obtiene una gran ventaja competitiva con respecto al resto de empresas que a lo mejor no pudieron implementar DevOps.

Barclays confía plenamente en esta cultura, tanto así que el ingeniero líder en DevOps de Barclays, Fletcher (2017) dijo “Creo que los profesionales del sector tecnológico han adoptado una actitud intrínseca para generar mejoras y lograr resultados reales. Desde el punto de vista de la

automatización, particularmente, cuando muestras a un colega cómo implementar un entorno de prueba en unos pocos pasos, no es necesario recurrir a las métricas, ya que sus rostros demuestran la felicidad que sienten al ver lo que la tecnología puede lograr". Barclays también incorporó en otras áreas de su empresa esta cultura como el área de seguridad, auditoría, esto se le conoce como DevOps integrales.

1.2.2 CASO DE ÉXITO DE NETFLIX

La automatización que ellos implementaron es a nivel de fallos, la única forma que ellos se sientan tranquilos con todo el sistema es lanzar fallos para aprender del problema y mejorar, el sistema es tan grande que pueden existir problemas y pueden ser lanzados de un rato al otro y además el número de microservicios son muchos. Entonces, desarrollaron una aplicación llamada Chaos Monkey que está escrita en Go[1], la función principal es detectar un grupo de instancias y detenerlos de manera automática y aleatoria, es muy probable que puedan existir nuevas fallas cuando se lanza un nuevo cambio para su sistema, por lo que ellos siguieren que un administrador del sistema analice y aprenda de este nuevo problema y posteriormente implementar la solución a dicha falla, de esta manera automatizar las fallas que se puedan dar a largo de todo su sistema distribuido el mismo que cuenta con varios componentes alojados en la nube de Amazon Web Service y desplegados en todas sus regiones y sobre todo ayudando a la mejora continua de su sistema y creando sistemas con tolerancias a fallos.

1.2.3 CASO DE ÉXITO DE ETSY

Etsy es una empresa de compra y venta de productos al estilo de Mercado Libre, Olx, entre otras, una empresa que por el año de 2008 tenía 35 empleados aproximadamente, la misma que contaba con el departamento de desarrollo encargado del escribir el código y operaciones que simplemente

[1] Go: Es un lenguaje de programación creador por Google, está pensado en dar soporte a múltiples propósitos

se encargaba de desplegar los nuevos cambios, el vicepresidente de operaciones técnicas Michael Rembetsy menciona que "las implementaciones a menudo son muy dolorosas, teníamos la mentalidad tradicional que los desarrolladores escriben el código y operaciones lo implementa. Y eso realmente no escala". Cambiaron la mentalidad de un enfoque tradicional en cascada a DevOps, pasaron de implementar funcionalidades dos veces por semana a ochenta al día, finalmente Michael Rembetsy (2018) dijo "se va a romper algo lo vamos a arreglar, si un servidor falla, lo consideraremos un éxito porque aprendimos algo nuevo", es algo similar al enfoque que Netflix tiene para el trato de sus fallos.

CapitalOne empresa de servicios financieros operando en Estados Unidos, es otro caso de éxito con la implementación de DevOps, para ellos DevOps se trata de "entregar trabajo de Software de alta calidad más rápido"(Pal, 2018), tratan de que sus fallos sean los mínimos posibles durante cada entrega, sin problema de seguridad, el cumplimiento de cada requerimiento, que cada componente funcione de manera correcta y la entrega continua de la manera más rápida sin perder calidad, de esta manera generar ese valor agregado. Antes de implementar DevOps, las entregas se hacían cada 3 meses, luego adoptar esta cultura, las entregas se hacen de manera diaria, semanal o por cada Sprint, alcanzando hasta 50000 procesos de construcción. Desde el momento que adoptaron DevOps, han desarrollado tres tipos de Pipelines, la primera que no la usan, ya que es una mala práctica de ramificación en la que existen ramas que son infinitas, la segunda trata de componentes que funcionan de manera independiente, cada desarrollador no tiene una rama si no que cuenta con su propio repositorio de código y la tercera en la que deben existir muchas personas para adminístralos, la misma tienen varios problemas como, por ejemplo, pruebas con errores, construcciones que no se ejecutan y cada uno arregla esos fallos. Pero si prestamos atención ninguna de estas tuberías tiene sentido para una empresa de tal magnitud más aun con lo que dijo

el director de Ingeniería de CapitalOne. Pal (2018) afirma que “todos los procesos implementados les permiten estar a la altura DevOps de ofrecer Software de trabajo de alta calidad sin comprometer ni sacrificar calidad o la velocidad de la canalización”. Entonces como es que trabajan, ellos diseñan su propio concepto de tuberías llamado “16 puertas”, que constan de buenas prácticas, como, por ejemplo, una buena control versiones, pruebas de rendimiento, análisis de vulnerabilidades, entre otros.

1.3 OBJETIVOS

1.3.1 OBJETIVO GENERAL

- Desarrollar e implementar un sistema basado en microservicios sobre una arquitectura DevOps para la automatización del proceso de despliegue e integración continua en infraestructuras basadas en código.

1.3.2 OBJETIVOS ESPECÍFICOS

- Revisar el estado del arte sobre tecnologías y/o herramientas de integración y despliegue continúo basada en contenedores.

- Definir la arquitectura de la plataforma mediante código que permita pasar del desarrollo a la implementación, en un entorno de producción en el que puede generar valor para la empresa.

- Desarrollar un sistema basado en microservicios y multiplataforma mediante tecnologías web.

- Implementar el sistema desarrollado dentro de la plataforma a fin de automatizar los procesos de prueba, validación y lanzamiento de la aplicación en un entorno de producción.

- Validar y probar el sistema dentro de la plataforma mediante métricas de rendimiento.

1.4 DEVOPS

Está definida básicamente por 2 palabras: desarrollo (Dev) y operaciones (Ops); es una metodología que describe formas para mejorar los procesos de una idea para pasar del desarrollo a la implementación dentro de un entorno de producción de esta forma generar un valor agregado para el cliente, disminuyendo tiempos de entrega, esta forma describe como el equipo de desarrollo y el de operaciones están relacionados y además deben contar con una buena comunicación. Así, DevOps es un movimiento cultural combinado con una serie de prácticas de desarrollo de software que permite un desarrollo rápido (Walls, 2015).

Dentro de DevOps, ambas partes tanto desarrollo y operaciones usarán herramientas en común. Por ejemplo, Docker soluciona una parte de su conflicto, hablarán un mismo idioma para evitar problemas de comunicación. Entonces, los desarrolladores crean contenedores dependiendo de la aplicación en sus máquinas locales y los ejecutan en el entorno de desarrollo, luego la parte de operaciones ejecuta el mismo contenedor en el entorno de producción (Robin, 2018). DevOps con Docker permite mejorar la velocidad con el flujo de desarrollo, flujo de integración y flujo de despliegue, permitiendo llegar a tener un mejor control sobre los cambios que se realizan dentro del proyecto.

Contextualizando de manera breve la implementación de DevOps sobre el proyecto actual, se pretende generar un flujo de procesos para completar la construcción o compilación de la aplicación que posterior será alojada en un repositorio de imágenes de manera versionada y

finalmente ser desplegada en un entorno de desarrollo, existen varios pasos para completar estas tareas los mismo que se explican de manera resumida a lo largo del presente capitulo.

1.4.1 COMPOSICIÓN DEVOPS

Consta de 2 partes y cada una de ellas inicia con ciertos procesos, que cuando se unifican forma todo el ambiente. (RedHat, 2020) Para la parte de Desarrollo (Dev) tenemos lo siguiente:

- Plan: contiene los diagramas iniciales y herramientas en las que se definen cada punto a desarrollar, por ejemplo, diagrama de clases, arquitectura, funcionalidades de la aplicación.
- Code: se especifica la herramienta de control de versiones que se usará para el proyecto y también la herramienta en la que se escribirá el código, como aplicación de control de versión se maneja Git para el proyecto.
- Build: es la construcción de la aplicación como tal o también conocida como fase de compilación, aquí depende del desarrollador que herramienta vaya a usar para compilar para que no existan errores de sintaxis, para la construcción de la aplicación se utiliza Docker para generar imágenes versionadas para cada una de las aplicaciones.
- Test: se definen las herramientas para realizar los diferentes tipos de prueba; por ejemplo, carga, unitarios, entre otras más.
- Release: es el lanzamiento de la aplicación de manera versionada, no es el despliegue, se debe tener en cuenta que son dos cosas totalmente diferentes, en esta parte la aplicación va a alojada a un repositorio de imágenes construida como imagen personalizada, para luego ser desplegada.

- Deployment: se procede a desplegar la aplicación en un entorno de desarrollo o de producción para que los clientes hagan uso de las nuevas funcionalidades, se utiliza como entorno de producción Kubernetes alojado en Digital Ocean como proveedor Cloud.
- Operate: se define una herramienta con la que se orquestará las aplicaciones y el número despliegues, como orquestador se hace uso de Kubernetes, para controlar las réplicas, los balanceadores de carga y servicios.
- Monitor: Es la parte final del flujo DevOps, define y utiliza herramientas que ayudan a monitorear el entorno de las aplicaciones de manera visual, para el proyecto se hace uso de Grafana para monitorear los recursos del Clúster.

1.5 CONTENEDORES

Es una plataforma que ayuda a los desarrolladores a crear, probar e implementar aplicaciones de una manera rápida, lo innovador es la velocidad con la que se puede implementar aplicaciones. Los contendores empaquetan cualquier tipo de aplicación con todas las herramientas que se necesiten para que pueda funcionar dicha aplicación, como el código, librerías y dependencias. El contenedor se podrá ejecutar en cualquier otra arquitectura de manera independientemente de cómo esté construida y donde se haya construido, esto es totalmente transparente para el desarrollador, como motor de contendores de utiliza Docker para la administración de contenedores.

1.5.1 DOCKER

Es una plataforma de código abierto escrito en el lenguaje de programación Go para desarrollar, probar y ejecutar aplicaciones. Docker separa las aplicaciones de la infraestructura, con el objetivo de entregar aplicaciones en la menor cantidad de tiempo posible. Docker empaqueta las aplicaciones y las ejecuta de manera aislada, lo que permite ejecutar varias aplicaciones de manera

simultánea, exponiendo las aplicaciones por medio de puertos. La comunicación principal entre el Daemon y el cliente se da a través de su Api [2] Rest. El Daemon [3]de Docker está al pendiente del Api para ejecutar el comando que el cliente solicita, el cliente simplemente envía los comandos que sean necesarios para levantar un contenedor. Por ejemplo, un contenedor con una base de datos o una aplicación web y finalmente el registro hace referencia al repositorio de las imágenes, en este caso el repositorio oficial para Docker es DockerHub, pero se puede subir imágenes a otros repositorios. (Docker, 2020)

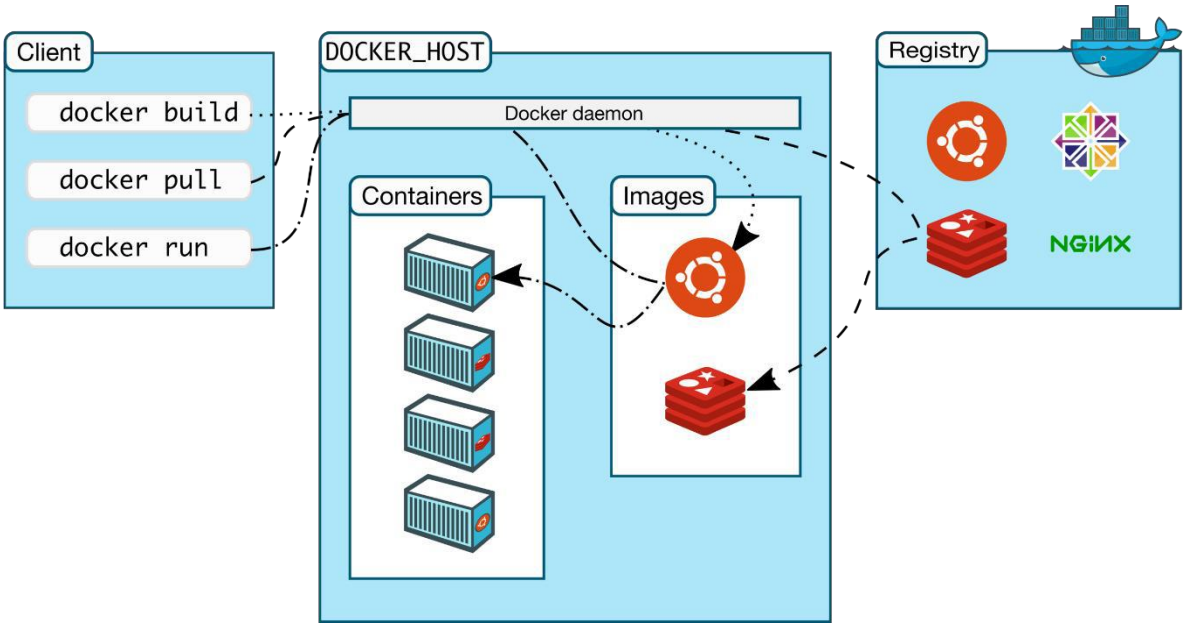


Ilustración 1 Arquitectura Docker. (Docker, 2020)

La arquitectura principal de Docker es cliente-servidor, para la comunicación entre los comandos el cliente interactúa con el Daemon de Docker, el objetivo principal es ejecutar los contenedores, exponer los puertos, crear volúmenes, para que la aplicación funcione de manera correcta. (Docker, 2020).

[2] Api: Interfaz de programación de aplicaciones, permite la comunicación con otras aplicaciones y el intercambio de información.
[3] Daemon: Conocido como demonio es un proceso que este asociado a algún tipo de servicio que se ejecuta en Background.

1.5.2 FUNCIONAMIENTO BÁSICO

Docker utiliza los recursos que hayan sido asignados o simplemente se asignan recursos de manera automática dependiendo de la memoria y procesador que disponga el servidor.

- Kernel: es la capa del SO, en la cual se ejecutará Docker y su demonio que lo administra, a diferencia de las máquinas virtuales esta se ejecuta directamente y sin esa capa extra que es la del hipervisor.
- Contenedores: Conjunto de aplicaciones encapsuladas o aisladas que se ejecutan dentro de Docker y sobre el Kernel, trabajan de manera independiente. (Redhat, 2020)

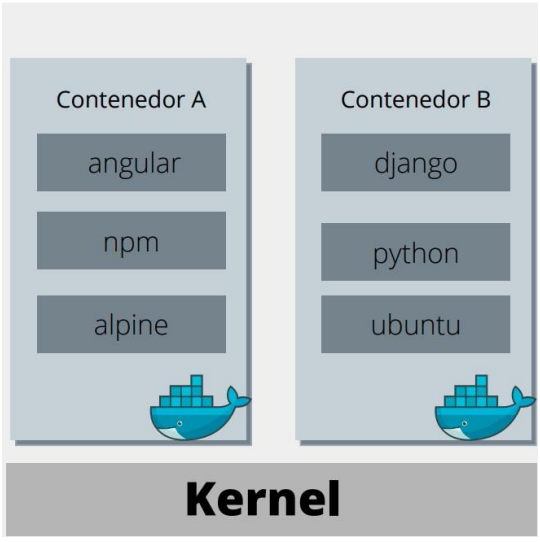


Ilustración 2 Ejemplo de dos contenedores Docker

1.5.3 CARACTERÍSTICAS

Docker posee varias bondades al momento de utilizarse. Por ejemplo, la portabilidad de un contenedor es simplemente fácil, ya que con un simple comando se puede compartir a cualquier persona o mover entre entornos de desarrollo, no importa en qué servidor o máquina se vaya a ejecutar, puesto que Docker está presente en varias plataformas, otra característica es la ligereza, en comparación con una máquina virtual, el peso es sumamente inferior, dado que desde la plantilla

o imagen son muy livianas, si comparamos el sistema operativo de Ubuntu de una imagen contra un archivo .Iso, encontramos que la imagen tiene un peso de 74mb contra 1.02Gb del archivo .Iso. Esto pasa por que las imágenes contienen solamente lo necesario para que un sistema operativo base se ejecute, también es escalable, aunque se debe contar con un soporte extra como lo es Docker Swarm[4].

1.5.4 DOCKERFILE

Es un archivo de configuración para crear imágenes personalizadas, con aplicaciones propias. Dockerfile contiene por lo menos un sistema operativo base para su funcionamiento, además contiene pasos (Steps) para generar la imagen final, en los que se especifican el contenido de la imagen.

1.4.4.1 SINTAXIS DOCKERFILE

Tabla 1 Sintaxis Dockerfile

Comando	Realiza	Ejemplo
FROM	Se utiliza para especificar la imagen base que va a contener la nueva imagen, por ejemplo, Nginx, Ubuntu, Python.	FROM node:14-alpine
RUN	Ejecuta comandos dentro de la imagen, por ejemplo, instalar una dependía, instalar pip.	RUN npm install
CMD	Ejecuta un comando al momento de iniciar el contenedor.	CMD ["npm", "start”]

[4] Docker Swarm: Es una herramienta que permite la administración de múltiples contenedores que pueden ser desplegados en varios nodos físicos o virtuales.

ENTRYPOINT	Es similar a CMD, con la diferencia que permite usar parámetros.	
WORKDIR	Define el directorio de trabajo, sería un equivalente a "cd /path", comando de Linux.	WORKDIR /usr/src/app
COPY	Copiar un contenido dentro de la imagen, por ejemplo, copiar un proyecto en Angular, librerías, entre otros.	COPY ./src ./src
ADD	Es similar a COPY, con la diferencia que permite copiar desde una Url.	
EXPOSE	Exponer puertos de la imagen, por ejemplo, puerto de Nginx 80.	EXPOSE 5000
VOLUME	Permite enlazar volúmenes para persistir la información.	

1.5.5 DOCKER COMPOSE

Es una herramienta de Docker que permite definir y lanzar varias aplicaciones (servicios) de forma simultánea, además puede definir la imagen base que va a contener el servicio, exponer puertos, usar volúmenes, redes y la afinidad que tiene con respecto a otros servicios, es una herramienta ideal para generar microservicios[5].

[5] Microservicios: trata de una arquitectura que desacopla una aplicación en pequeños servicios independientes de esta manera poder llevar una mejor mantenibilidad, agregar nuevos cambios a futuro sin muchos problemas.

```yaml
version: "3.9"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Ilustración 3 Sintaxis docker-compose (Docker, 2020)

Tabla 2 Descripción docker-compose

version	Versión de archivo yaml, para la versión actual de Docker-Compose
services	Se definen cuantos servicios va a tener el manifiesto.
db	En este caso es un nombre que se le da al servicio.
image	Se especifica el nombre de la imagen base, para el servicio db.
enviroment	Se especifican las variables de entorno en el caso que el contenedor necesite.
build	Se tomará un archivo Dockerfile para construir una imagen.
command	Comandos que se ejecutan dentro del contenedor.

volumes	Se define un volumen para persistir o compartir información.
ports	Se definen puertos para mapear con respecto a puertos internos del Conteiner.
depends_on	Depende de otro servicio para poder ejecutar el servicio actual.

1.6 KUBERNETES

Es una herramienta de código abierto que permite administrar, desplegar y automatizar aplicaciones. A comparación con Docker, Kubernetes lo hace todo a gran escala, cada nodo del Clúster permite tener hasta 110 Pods de una misma aplicación.

Kubernetes permite todo esto por medio de la configuración declarativa, especificar como va a estar definido un despliegue. Por ejemplo, el número de réplicas, que puertos expone el contenedor, si se tienen variables de entorno, manejo de volúmenes, que recursos de memoria y CPU tiene cada replica, para el presente proyecto, se han definido varias características de forma declarativa, como lo son los puertos que se hacen uso, por ejemplo, 8080, 80 y demás, se hace uso de volúmenes para persistir la información en caso de que ocurra problemas con el Pod de la base de datos.

1.6.1 CLÚSTER KUBERNETES

El Clúster de Kuberentes es la composición de varios nodos, el principal es el plano de control y el resto son nodos Workers.

1.6.1.1 NODO MÁSTER

También conocido como plano de control este nodo es el núcleo de todo el Clúster, ya que tiene como principal objetivo mantener un estado deseado del mismo y también lleva el control de

manera minuciosa de cada nodo Worker, es tan preciso que sabe cuándo un Worker está caído. Además, asigna tareas al resto de Workers, por ejemplo, levantar nuevos Pods, decide a que nodo va cada Pod.

En el nodo principal siempre se ejecutarán tres procesos que son exclusivos del Nodo Máster los cuales son:

- Kube-apiserver: permite la configuración y validación de los objetos del Clúster de Kubernetes, se puede manipular para realizar cambios al Clúster, pero no es recomendable porque esta herramienta configura y valida todo automáticamente.
- Kube-controller-manager: es un bucle de controles centrales que se envían al Clúster, este bucle es sin terminación que ayuda a regular el sistema.
- Kube-scheduler: es el planificador que de manera automática asignan las cargas de trabajo a los Pods de los nodos existentes dentro del Clúster.

1.6.1.2 NODO WORKER

El nodo Worker es quien se encarga de ejecutar los Pods que son asignados por el nodo Master.
En los Nodos Workers se ejecutan 2 procesos:

- Kubelet: permite la comunicación contra el Nodo Máster.
- Kube-proxy: está asociado a los servicios y permiten la comunicación interna o externa hacia el Backend asociado al servicio.

1.6.2 OBJETOS KUBERNETES

Kubernetes se compone de varios objetos que son abstracciones que representan una parte del Clúster, como recursos de red, espacios de nombres, Pods, Deployments, Services, entre otros, de

igual manera dentro del proyecto se han definido varios objetos que tienen un propósito bien definido, como puede ser el que permite exponer los servicios por lo que acceden los clientes, para la comunicación, todos los procesos se detallan en los siguientes capítulos.

1.6.2.1 PODS

Son la unidad mínima de trabajo que existe dentro de Kubernetes, básicamente es cualquier aplicación dentro de un contenedor, pueden existir varias réplicas de los Pods, con un máximo de 110 por cada nodo, comparten recursos, como almacenamiento, red, además se pueden limitar en uso de recursos como memoria y CPU.

Los Pods son efímeros, esto quiere decir que no es seguro que una aplicación dentro de un Pod vaya a estar siempre disponible y para solucionar este problema existen otros objetos que ayudarán a mantener disponible la aplicación.

1.6.2.2 DEPLOYMENT

Es una forma de declarar una aplicación, pero con ciertas características adicionales, dentro de ellas permite definir el número de réplicas que la aplicación tendrá. Entonces, si en algún momento del tiempo se llega a borrar un Pod de los n Pods que hayan sido definidos, automáticamente se levantará una nueva instancia del Pod y de esta manera siempre estar dentro del estado deseado.

1.6.2.3 SERVICE

Es una forma de exponer las aplicaciones que están dentro de los Pods, no es necesario agregar una configuración extra para la comunicación entre los Pods pues estos comparten recursos, se asignan Ip’s de manera dinámica con un nombre de dominio (Dns) entre todas las réplicas (Pods), de esta manera se equilibra la carga, entonces el servicio no apuntará a todos los Pods lo que hace es apuntar al Deployment que contiene todos estos Pods a través de selectores.

1.6.2.3.1 TIPOS DE SERVICIOS

Para acceder a las aplicaciones que se han definido en los Deployments se lo hace a través de servicios, Kubernetes dispone de varios tipos cada uno con su diferenciador.

- ClusterIP: exponer el Deployment con una Ip interna del Clúster, es decir que es solo visible dentro del Clúster, normalmente se utilizan para conexiones internas.
- NodePort: expone el Deployment asignándole un puerto por el que los clientes podrán acceder, por defecto NodePort tiene un rango de puertos que van desde el 30000 hasta 32767, de esta manera se pude acceder por medio de la Ip de algún nodo del Clúster y su puerto asignado.
- LoadBalancer: expone el Deployment de manera externa mediante un equilibrador de carga, esto depende del proveedor en el cual el Clúster esté desplegado, se generará un balanceador de manera automática, se pude asignar un puerto aleatorio o por defecto es el puerto 80.
- ExternalName: Asigna el servicio a un nombre de Dns, al devolver un registro CNAME con su valor.

1.6.2.4 VOLÚMENES

Al igual que los Pods, los espacios en disco dentro del Pod también son efímeros, por tal motivo se hacen uso de los volúmenes para persistir la información, de manera automática los volúmenes se asignan a los Pods. Una ventaja es la posibilidad de implementar un volumen externo al Clúster, dependiendo del proveedor se recomienda usar uno propio o tener un servidor Nfs [6].

[6] Nfs: Sistema de archivos en red, utilizado para el almacenamiento de archivos.

1.6.2.5 NAMESPACES

Son Clústeres pequeños virtuales dentro del Clúster físico, se conocen como espacios de nombres, dentro de cualquiera se puede hacer Deployments, Services y otros. Por defecto, dentro del Clúster existen cuatro Namespaces:

- default: Espacio predeterminado para lanzar aplicaciones, se pueden crear otros.
- kube-system: Espacio para los objetos que son propios de Kubernetes.
- kube-public: Espacio reservado para el uso del Clúster.

kube-node-lease: Espacio predeterminado para los objetos de arrendamiento asociado a cada nodo.

1.6.2.6 REPLICASET

Trabaja de manera simultánea con los Pods para mantener un estado estable y así garantizar disponibilidad de la aplicación asociada. Cada vez que exista un cambio en el Deployment el ReplicaSet estará pendiente si el nuevo estado deseado tiene más replicas o menos, si es el caso de tener más replicas creara nuevos Pods, caso contrario eliminara los Pods que sobren.

1.6.2.7 DAEMONSET

Tiene como principal objetivo garantizar que todos los nodos del Clúster o la mayoría ejecuten una réplica de un Pod en caso de tener varias réplicas y más de un nodo Worker.

1.6.3 K8S

Originalmente desarrollada por Google y liberada en 2015, escrita en el lenguaje de programación Go y de código libre. Es un orquestador a gran escala para administrar contendores Docker. K8s es la versión de Kubernetes que generalmente se usa para producción, se lo instala directamente en el Hardware, los proveedores Cloud usan para definir los Clúster. Permite la integración con

múltiples aplicaciones. Por ejemplo, aplicaciones para el monitoreo de los objetos, aplicaciones para ver métricas de Clúster, la migración de microservicios de docker-compose a Kubernetes (Kubernetes, 2020).

1.6.4 K3S

Es una versión de Kubernetes más liviana a K8s, pero que no está creada para un entorno de producción. Pensada para los entornos que no tiene muchos recursos, por ejemplo, un Raspberry Pi.

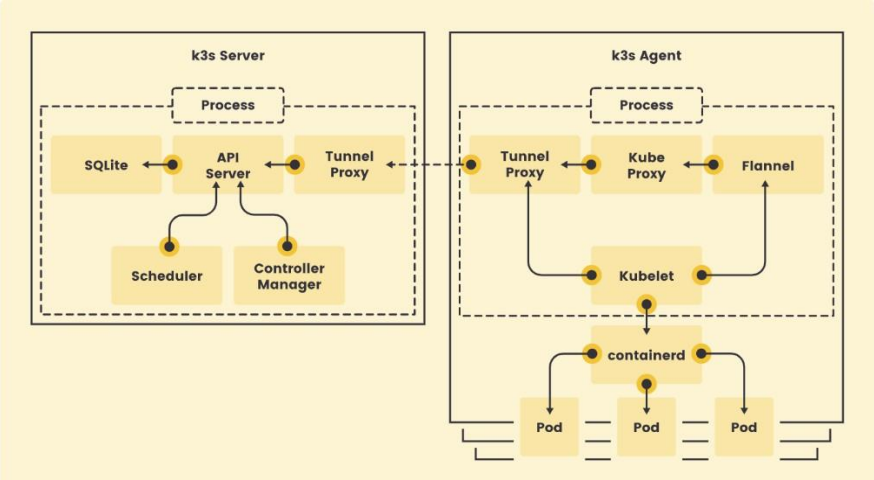


Ilustración 4 Arquitectura K3s (K3s, 2020)

Sus principales usos son para:

- Iot
- CI
- Desarrollo

Características principales:

- Pensado para entornos ligeros.

- Soporta Sqlite, MySQL y Postgres para el almacenamiento de parámetros predeterminados.

1.6.5 LET'S ENCRYPT

Actualmente es una herramienta que provee certificados Ssl, además es gratis y los certificados son generados de manera automática, auto firmados y con renovado automático, evitando el proceso tradicional para generar certificados manuales, todo esto se logra gracias al protocolo ACME, el cual gestiona la validación y la instalación del certificado en el servidor sin la necesidad de realizar pasos extra. También, Let's Encrypt cuenta con un servidor para realizar pruebas y otro servidor de producción, lo necesario para los certificados son tener un Dns[7] y un servidor Web Nginx (Lets Encrypt, 2020).

1.6.6 CERT-MANAGER

Es una herramienta que trabaja a la par con Let's Encrypt, el cual gestiona la generación de certificados que apuntan a un dominio o subdominio, teniendo un único punto de entrada hacia las aplicaciones que se hayan definido.

1.6.7 NGINX

Es un servidor Web que tiene una gran capacidad para recibir las peticiones y reenviar a los servicios configurados, además funciona como un LoadBalancer independiente o como un servicio de tipo LoadBalancer que se configura dentro de Kubernetes.

[7] Dns: es el sistema de nombre de dominio que permite conectarse a un servicio a través de un nombre antes que utilizar una dirección Ip, por ejemplo, www.ups.edu.ec.

1.6.8 INGRESS

Es un objeto de Kubernetes que permite la comunicación de las aplicaciones hacia el exterior de Clúster, con grandes diferencias a comparación de los servicios de Kubernetes que también exponen las aplicaciones al exterior. A continuación, se describen las diferencias:

- El Ingress se conecta a un servicio de tipo ClusterIP
- Balancear el tráfico de los servicios a través de un único punto.
- Permite el acceso mediante los protocolos Http y Https.
- Ideal para exponer aplicaciones Web y balanceo de cargas.

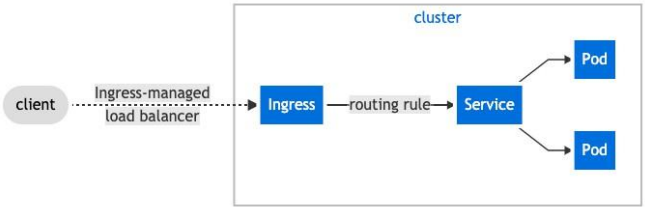


Ilustración 5 Ingress (Kubernetes, 2020)

1.6.9 GRAFANA Y PROMETHEUS

Grafana es una herramienta de código abierto que ayuda a mostrar datos y crear cuadros de mando que permiten ver información acerca del estado de un servicio, métricas de rendimiento, como CPU, memoria, mientras que Prometheus es una herramienta de código abierto que tiene la ventaja de monitorear, capturar y alertar sobre cualquier problema basado en las métricas propias de un servidor o en este caso de un Clúster de Kubernetes (Chawla y Kathuria, 2019).

La relación que une Grafana con Prometheus es a través de Datasources, el cual tiene que ser expuesto a través de una dirección, para el presente propósito Prometheus estará expuesto por un servicio de tipo Cluster Ip, todas las métricas que provee Prometheus serán accedidas por el servicio, el mismo será usando en Grafana para su posterior conexión y uso.

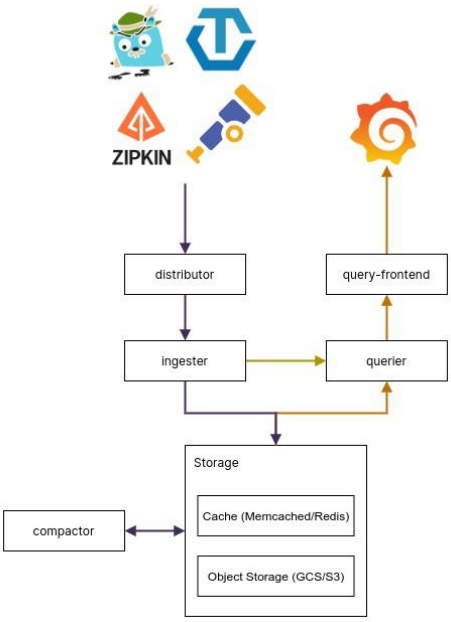


Ilustración 6 Arquitectura Grafana (Grafana, 2020)

Entonces, Grafana permite visualizar las métricas que expone Prometheus a través de múltiples gráficos que soporta. Entre las características más importantes que se tienen son:

- Permite la conexión a múltiples fuentes de datos.
- Diferentes tipos de gráficos como histogramas, de calor, entre otros.
- Consultas en tiempo real y en diferentes espacios de tiempo.
- Creación de paneles propios.

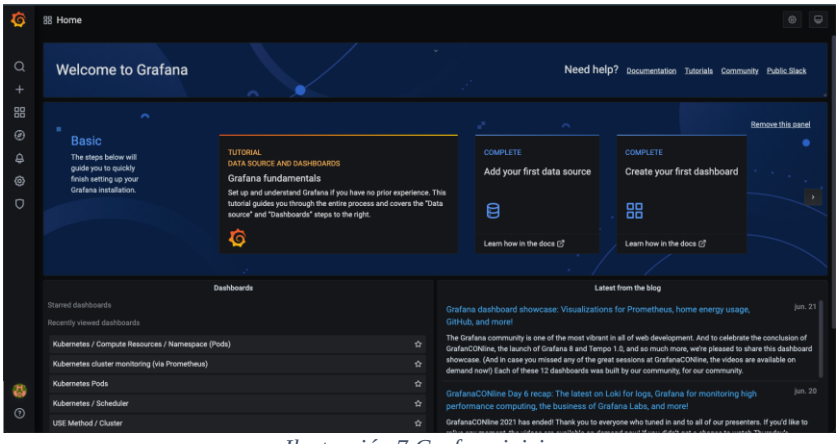


Ilustración 7 Grafana inicio

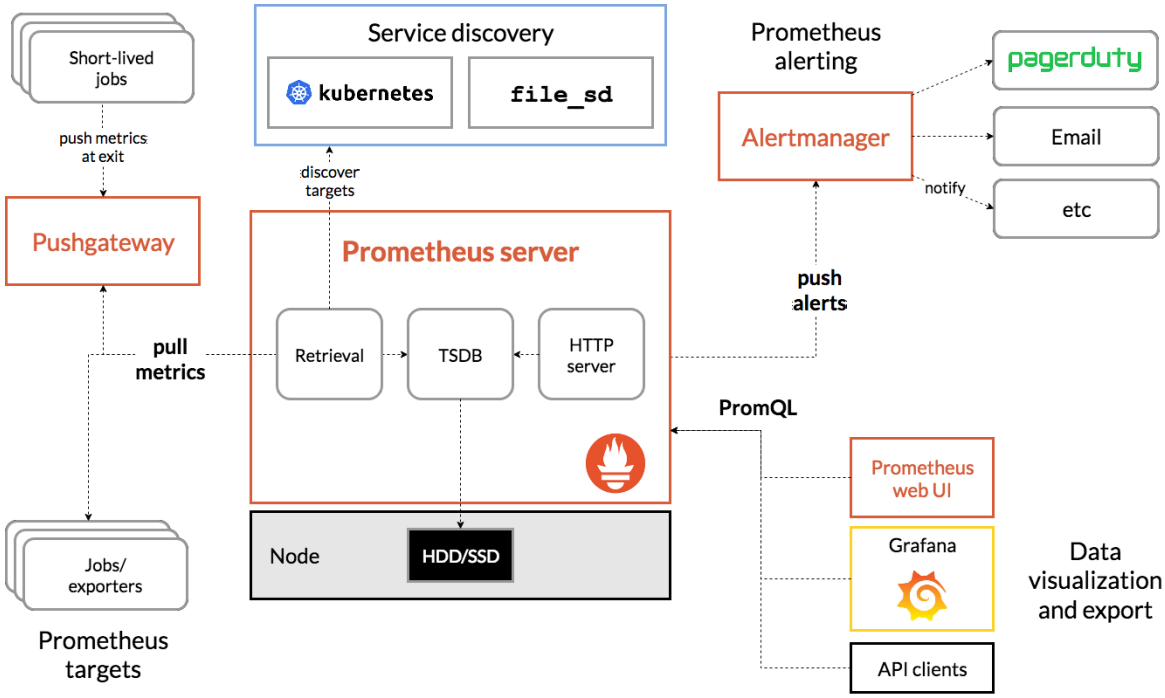


Ilustración 8 Arquitectura Prometheus (Prometheus, 2020)

Como tal Prometheus es una herramienta que principalmente funciona para monitorear recursos, además cuenta con un sistema de alertas llamado AlertManager que funciona en base a métricas de rendimiento. Por ejemplo, si existe sobrecarga en memoria, Cpu, o incluso cuando un contenedor deja de funcionar, AlertManager envía notificaciones describiendo el problema según la métrica, puede ser enviado al correo o algún canal de Slack. Además, Prometheus cuenta con algunas características como:

- Lenguaje de consulta flexible PromQL.
- Dispone de métricas definidas para múltiples necesidades.
- Soporta varias ventanas de gráficos.

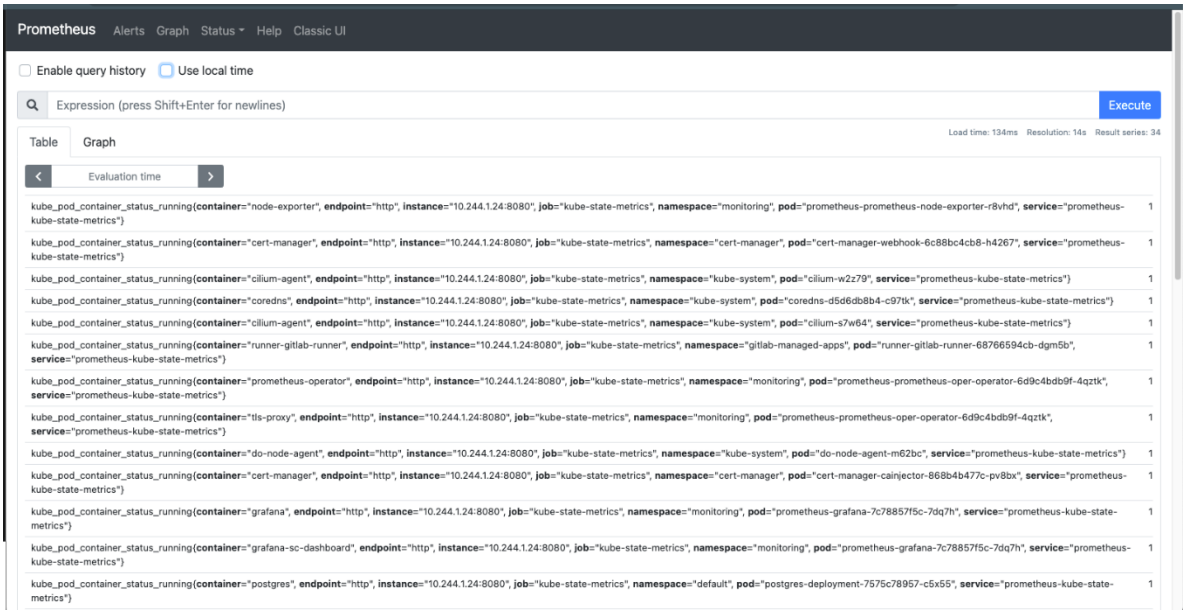


Ilustración 9 Tablero Prometheus

1.7 INTEGRACIÓN Y DESPLIEGUE CONTINUOS

Para completar el proceso de integración y despliegue continuo hay que considerar que herramientas son las adecuadas para realizar toda la canalización con sus respectivos procesos, a continuación, se analizan de manera breve algunas de las herramientas que permiten definir un flujo de trabajo para completar la integración y despliegue continuo, algunas solamente permiten realizar integración continua y otras permiten realizar ambos procesos. Dependiendo de las necesidades se debería optar por una y otra, por lo que es importante analizar cada una de ellas con más profundidad antes de seleccionar una herramienta.

1.7.1 INTEGRACIÓN CONTINUA (CI)

Principalmente dentro de CI trata de integrar todo el código de las personas que trabajan en un mismo proyecto, generar un archivo binario, un ejecutable, o en este caso generar una imagen con todo lo necesario para que funcione la aplicación, este proceso es previo al despliegue de la aplicación a un entorno de producción o desarrollo.

Trabaja con un repositorio de código y de imágenes, en los cuales se realizan 3 procesos, build o construcción de la aplicación, test automatizados y Merge hacia un repositorio, durante la ejecución de estos tres procesos se toma en cuenta que el código no rompa partes de la aplicación o evite conflictos con el código de los demás, en este punto se toman medidas necesarias para capturar este tipo de excepciones y notificar a cada desarrollador, al final de todo esto, si pasa cada etapa con éxito. Por ejemplo, se debe subir la imagen con la aplicación lista para ser lanzada a producción.

Existen varias herramientas con las cuales se pude tener un entorno para realizar integración continua, por ejemplo, Jenkins, Travis Ci, Gitlab Ci. Cada una tienen diferentes características, unas más manuales y otras automáticas.

La forma de implementar la parte de integración continua está pensada con Docker para construir la imagen que contiene la aplicación con todas las herramientas necesarias de esta forma es más fácil subirla a un repositorio de imágenes y con Git y Gitlab se versionará de manera adecuada cada cambio que se realice dentro de la aplicación.

1.7.2 DESPLIEGUE CONTINÚO (CD)

Entrega continúa, despliegue continúo, también se conoce como distribución continúa, se encarga de tomar la imagen o binario que está en el repositorio y lanzarlo a producción o a algún servidor de desarrollo, se debe tener en cuenta otros puntos, como por ejemplo, que la aplicación se actualice con los nuevos cambios y para ello se debe utilizar un Tag que generalmente es el Commit Sha que provee el propio Git o simplemente generar un Tag único por cada cambio que se lo genera en la parte de integración continua.

1.7.3 JENKINS

Servidor para la automatización de código, es de código abierto, permite realizar todo tipo de pruebas, desarrollo y entrega de aplicaciones, se puede instalar desde archivos binarios, además se puede ejecutar en un entorno de contenedores. (Jenkins, 2020)

CARACTERÍSTICAS JENKINS

- Lanzado en 2011.
- Desarrollado en Java.
- Multiplataforma.
- Tiene soporte para CD.
- Tiene un api para el control, línea de comandos y GUI.
- Código abierto.
- Tiene más de 1400 Plugins para automatizar procesos.
- Tiene una arquitectura distribuida controlada por un nodo Main.

1.7.4 TRAVIS CI

Está enfocado al CI, tiene una similitud a cómo trabaja GitHub por su sistema de versionado, como característica funciona en GitHub, ya que a través de archivos Yaml, GitHub informa todos los cambios realizados en el repositorio. (Travis, 2020)

CARACTERÍSTICAS TRAVIS CI

- Lanzado en 2012.
- Desarrollado en Ruby.
- Multiplataforma.

- Funciona en GitHub.
- Código abierto.
- Funciona en base a manifiestos.
- Pasó a ser de pago a finales de 2020.
- Costo 64 y 489 dólares por mes.

1.7.5 BAMBOO

Herramienta que ayuda con CI y CD, se integra con los repositorios de Bitbucket, funciona a través de una interfaz gráfica.

CARACTERÍSTICAS BAMBOO

- Lanzado en 2007.
- Desarrollada en Java.
- Multiplataforma.
- Se integra con Bitbucket.
- Gratis para proyectos de código abierto.
- Costo entre 10 y 126500 dólares, depende del número de servidores a pago único.

1.7.6 GITLAB CI/CD

Es una herramienta desarrollada por la empresa Gitlab, permite el desarrollo de software a través de metodologías continuas como lo son CI y CD (Gitlab, 2020).

CARACTERÍSTICAS GITLAB CI/CD

- Desarrollado en Ruby [8] y Go.
- Funciona en base a manifiestos.
- Versión gratuita con características limitadas.
- Hace uso de variables de entorno propias de Gitlab.
- Costo entre 4 y 99 dólares.

1.7.7 CIRCLE CI

Herramienta para integración continua funciona tanto en GitHub y Bitbucket, usan contenedores o máquinas virtuales.

CARACTERÍSTICAS CICLE CI

- Construye de manera automática para otros entornos.
- Multiplataforma.
- Soporta CD.
- Gratuita si se instala dentro de un contenedor.
- Costo entre 50 y 3150 dólares al mes.

1.7.8 CRUISE CONTROL

Es una herramienta pionera en lo es CI, que salió a la luz por primera vez en 2001 y actualmente continúa en desarrollo.

CARACTERÍSTICAS CRUISE CONTROL

- Escrito en Java.

[8] Ruby: Es un lenguaje de programación de código abierto es orientado a objetos creado por Yukihiro Matsumoto.

- Multiplataforma.
- Control por medio de interfaz gráfica en la web.
- Código abierto.
- Gratuita.

1.8 INFRAESTRUCTURA COMO CÓDIGO

Hace referencia a la práctica de automatizar la infraestructura a través de código, esto permite reducir tiempo para desplegar arquitecturas, permite automatizar la creación de arquitecturas en la nube, creación o asociación de claves Ssh, recursos como memoria, Cpu, y demás. Es muy común que los desarrolladores escriban sus propios Scripts para generar sus propias infraestructuras, dependiendo de las necesidades.

Una de las ventajas al momento de escribir una infraestructura es que se pueden reutilizar en varios proveedores de la nube o inclusive se puede generar en un entorno de pruebas usando virtualizadores. Existen varias herramientas que permiten generar infraestructuras, por ejemplo, Terraform, Puppet, Ansible, Chef.

1.8.1 TERRAFORM

Es una herramienta de código abierto, tiene su propio Cli, para la administración y generación de infraestructuras, permite generar claves Ssh, maquinas, registros Dns. Tiene soporte para varios proveedores Cloud como Google Cloud, Amazon Web Service, Azure, Digital Ocean, VMWare (Terraform, 2020).

```
resource "aws_instance" "iac_in_action" {
  ami               = var.ami_id
  instance_type     = var.instance_type
  availability_zone = var.availability_zone

  // dynamically retrieve SSH Key Name
  key_name = aws_key_pair.iac_in_action.key_name

  // dynamically set Security Group ID (firewall)
  vpc_security_group_ids = [aws_security_group.iac_in_action.id]

  tags = {
  Name = "Terraform-managed EC2 Instance for IaC in Action"
  }
}
```

Ilustración 10 Sintaxis Terraform (Terraform, 2020)

Terraform analiza lo que en el archivo de configuración este definido y posteriormente crear los recursos de Hardware y recursos necesarios para la administración de la infraestructura, todo esto dependerá del proveedor Cloud que se haya seleccionado.

1.8.2 PUPPET

Es una herramienta de código libre con una versión Enterprise, permite configurar y automatizar servidores, tiene su propia arquitectura servidor-agente.

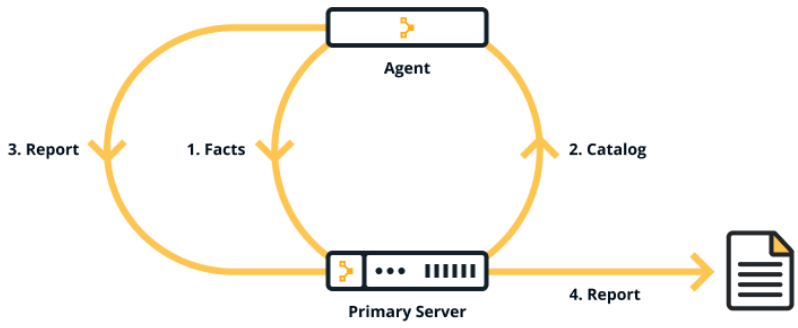


Ilustración 11 Arquitectura servidor agente (Puppet, 2020)

1.8.3 ANSIBLE

Herramienta de automatización y configuración de infraestructura, que tiene una particular característica de permite orquestar procesos como CI.

1.9 RANCHER

Es una herramienta de código abierto, que permite la orquestación de contenedores y generación de Clúster de Kubernetes además de su respectiva administración, todo esto en producción o desarrollo, para Kubernetes permite el escalado de aplicaciones y la administración de cada objeto que posee el Clúster, además tiene soporte para varios proveedores Cloud y fácil despliegue con Tokens de acceso.

1.10 GITLAB

Plataforma web para el trabajo colaborativo en desarrollo de Software y control de versiones de manera gratuita y con versiones Enterprise, además permite realizar CI y CD con conexión directa al Clúster de Kubernetes u otros entornos. Gitlab está encargado de alojar el código fuente de las aplicaciones para realizar posteriormente lo que es la integración y el despliegue continuos.

1.10.1 VARIABLES DE ENTORNO CON GITLAB

Las variables de entorno que brinda Gitlab son exactamente iguales a las que se manejan en los sistemas operativos que normalmente son usadas, se puede crear de forma manual o usar las que tiene por defecto como por ejemplo, la variable de entorno que tiene el id de un Commit, las variables de entorno que se crean se reflejarán únicamente dentro del proyecto, las variables se usarán dentro del archivo gitlab-ci.yaml, más que nada para almacenar credenciales y nombres que se les dará a las imágenes Docker, todo esto para evitar la exposición de credenciales en texto

plano y como una buena práctica, en el que se especifica pasos a seguir para realizar la tarea de construcción de la aplicación y su posterior despliegue.

1.10.2 GITLAB CI CD

Esta herramienta que permite definir pasos para realizar el proceso de integración y despliegue continuos, todos los pasos son definidos dentro de un archivo gitlab-ci.yaml, que contiene las instrucciones necesarias para cumplir con todos las tareas de automatización, permite usar máquinas por defecto que Gitlab brinda o en su defecto permite hacer uso de un Clúster de Kubernetes que debe ser previamente configurado con los agentes de Gitlab también llamados Runners.

1.10.3 RUNNERS

En español conocido como corredores, son agentes propios de Gitlab que su labor será realizar el proceso de ejecución de los pasos que se especifican en el archivo gitlab-ci.yaml, los Runners se pueden instalar en un entorno externo o utilizar los que brindan propiamente Gitlab, pero con esto no se tiene un control, ya que se crea una máquina independiente para ejecutarse, la opción más conveniente es instalar de manera automática o manual uno dentro de un entorno propio para tener control, aunque los Runners son livianos y escalables, por defecto el proyecto hará uso de los Runners que Gitlab provee, pero se puede cambiar por uno que se haya instalado dentro de un Clúster (Gitlab, 2020).

1.10.3.1 CARACTERISTICAS RUNNERS

Pueden ser de tres tipos, compartido, grupal o específico; el compartido, se puede usar por varios proyectos, el grupal se usa para subgrupos de proyectos y el específico es un Runner que se le instala dentro de un entorno propio para más control.

- Token: permite la comunicación del Runner hacia el proyecto, puede ser un Runner específico o compartido.
- Descripción: información adicional del Runner.
- Versión: versión del Runner.
- Ip: dirección del host en el que se instaló el Runner.
- Proyectos: en qué proyectos o proyecto está ejecutándose.

1.10.4 FICHERO GITLAB-CI.YAML

Este archivo dependerá del lenguaje de programación o entorno en que se desee ejecutar los pasos para integración y despliegue continuos, el archivo contiene pasos para cumplir con determinadas tareas, dichos pasos se ejecutan en una máquina propia de Gitlab o en un entorno propio.

Tabla 3 Manifiesto Python (Gitlab, 2020)

```
variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"
cache:
  paths:
    - .cache/pip
    - venv/

before_script:
  - python -V  # Print out python version for debugging
  - pip install virtualenv
  - virtualenv venv
  - source venv/bin/activate

test:
  script:
    - python setup.py test
    - pip install tox flake8  # you can also use tox
    - tox -e py36,flake8

run:
  script:
    - python setup.py bdist_wheel
    - pip install dist/*
  artifacts:
    paths:
      - dist/*.whl

pages:
```

```
  script:
    - pip install sphinx sphinx-rtd-theme
    - cd doc ; make html
    - mv build/html/ ../public/
  artifacts:
    paths:
      - public
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

Principalmente lo que se describe dentro de un manifiesto son los pasos por seguir para cumplir una determinada tarea, dependiendo del lenguaje utilizado o herramienta, por ejemplo, en la tabla 3, Gitlab define un manifiesto para que un servicio con Django pueda ejecutar los Test que se han definido, según la necesidad dentro del manifiesto incluso puede llegar a configurar conexiones Ssh hasta realizar despliegues de manera automática.

CAPÍTULO 2 ANÁLISIS, REQUERIMIENTOS Y PLANTEAMIENTO DEL PROBLEMA

2.1 PROBLEMA DE ESTUDIO

Actualmente, la construcción y despliegue de las aplicaciones toma un tiempo enviar a producción ya que todo el proceso es manual, es manual en el sentido que todo el proceso lo hace una persona cuando hay un cambio dentro del proyecto o cuando se lanza una aplicación por primera vez, se construye un binario o ejecutable a mano y se lo lleva al servidor para luego ser desplegada en un ambiente de producción o desarrollo, todo esto conlleva la pérdida de tiempo, dinero y además quita ese valor agregado al cliente. La construcción de la aplicación bien sea el binario, un ejecutable o para este caso que se maneja un entorno de contendores, la imagen que contiene la aplicación puede presentar problemas al momento de ejecutar, ya sea por un error de compilación o un error generado de manera involuntaria por alguna persona del equipo. Para evitar los diferentes problemas que surgen al momento de lanzar un cambio en la aplicación o mejorar el tiempo de entrega de una aplicación, existen diversas formas de solucionar estos problemas, como, por ejemplo, la implementación de DevOps como una metodología de desarrollo que se asocia a metodologías agiles permitiendo llevar de mejor manera el desarrollo de un proyecto desde el inicio hasta lo que es el despliegue, la misma que se aplica en el proyecto.

2.2 JUSTIFICACIÓN

Son múltiples escenarios donde se puede implementar DevOps no importa el ámbito o si empresa pequeña o grande, ya que DevOps no trata de una herramienta sino más bien de una metodología en el que se hacen uso de buenas prácticas para el desarrollo e implementación de proyectos conjuntamente con la automatización sin importar que lenguaje o tecnología se maneje, puesto que si se habla de contenedores Docker tiene una imagen lista para cada tecnología, lo único que se

debería implementar es justamente la parte que se piensa desarrollar, Docker cuenta con imágenes, como, por ejemplo, para Python, Java, Php, bases de datos, servidores de aplicaciones, servidores Web, entre muchas más. Aun mejor si por algún motivo no se llegara a confiar en alguna imagen que Docker provee se puede crear una imagen propia totalmente desde cero que solamente contenga lo que cada uno necesite. Además, con Kubernetes se puede administrar toda una arquitectura que contenga balanceadores de carga, certificados seguros que sean propios o de terceros, servidores Web, entre otras más, gracias al uso de manifiestos declarativos y todo esto puede ser administrado a gran escala.

La intención que se tiene con el presente proyecto es dar una solución a múltiples empresas que desarrollan su propio Software o que los comercializan, una solución que va a evitar que los equipos de trabajo pierdan mucho tiempo desde que realizan los diferentes tipos de prueba hasta el despliegue y esto lleva a que no pueden entregar un valor agregado a sus clientes, ni mucho menos software que cumpla con los requerimientos planteados. Desde el punto de vista de los desarrolladores por el lado del desarrollo(Dev) y el de operaciones (Ops), estos equipos de trabajo no tienen un entorno en el que puedan trabajar de manera transparente, ya que todo lo realizan de forma manual, para solucionar esta problemática se propone, desarrollar e implementar una arquitectura implementando DevOps que cualquier empresa de diferentes servicios podrá solucionar estos problemas al momento de empezar un nuevo proyecto, de esta forma logra que la empresa pueda evitar pérdidas económicas y de talento humano, generando una grupo de trabajo eficiente y con proyectos eficientes a largo y corto plazo.

La empresa gana tiempo en cuestiones de entrega y soluciones de problemas, brindando un valor agregado a todos sus clientes, cumpliendo requisitos y sobre todo una mejor planificación. Permite

a los desarrolladores tener un entorno transparente para que puedan realizar su trabajo y cumplir todas las pautas planteadas dentro de un proyecto.

2.3 ELICITACIÓN DE REQUERIMIENTOS

2.3.1 REQUERIMIENTOS

Teniendo en cuenta que todo sistema, arquitectura o implementación debe tener un punto de partida como son los requerimientos que deben contar con "una descripción de una condición o capacidad que debe cumplir un sistema, arquitectura o implementación, ya sea derivada de una necesidad de usuario identificada, o bien, estipulada en un contrato, estándar, especificación u otro documento formalmente impuesto al inicio del proceso" (Chaves, 2005).

Principalmente los requerimientos se relacionan con respecto a la integración continua, entrega continua y la arquitectura, cada uno de los procesos tendrán relación entre la una y la otra, considerando el problema de estudio principal y las desventajas que se dan al momento de lanzar una nueva aplicación o realizar cambios que se han definido a través de varios requerimientos.

2.3.2 REQUERIMIENTOS FUNCIONALES

Se dice que "los requerimientos funcionales describen lo que el sistema debe hacer" (Sommerville, 2005), entonces estos requerimientos hacen referencia a las características que debe tener un sistema o arquitectura con respecto a la interacción con el usuario, por ende, se han definido los siguientes requerimientos:

Con respecto a la integración continua se tiene:

- Las pruebas definidas se deben ejecutar como primer punto antes de continuar con el flujo de trabajo, en caso de error simplemente no se ejecutan el resto de los procesos.
- El repositorio para cada imagen y sus respectivas versiones se alojarán en DockerHub cada vez que un desarrollador realice un cambio en la rama principal.

- Para el caso del Backend se debe agregar la opción de generar el archivo jar dentro del Clúster para ser usado con Docker para la construcción de la imagen.
- Para el caso del Frontend se debe usar como imagen final Nginx, dado que el proyecto de Angular debe ser compilado a proyecto JavaScript.

Con respecto a la entrega continua se tiene:

- Simplemente se deberá actualizar los contenedores que contengan a la aplicación que se esté cambiando, no se debe agregar ninguna otra configuración adicional.

Con respecto a la arquitectura se tiene:

- Debe existir un único punto de entrada a los servicios a través de un balanceador de carga.
- Se deberá definir por una única vez la estructura que deben tener los contenedores a través de manifiestos.
- La base de datos estará atada a un volumen externo al Clúster para evitar perdida de información.

2.3.3 REQUERIMIENTOS NO FUNCIONALES

Los requerimientos no funcionales "son requerimientos que no se refieren directamente a las funciones específicas que proporciona el sistema, sino a las propiedades emergentes de éste como la fiabilidad, el tiempo de respuesta y la capacidad de almacenamiento" (Sommerville, 2005). En otras palabras, con las características del sistema o arquitectura que son totalmente invisibles para el usuario y trata principalmente del funcionamiento.

Con respecto a la arquitectura:

- El Clúster de Kubernetes debe tener como mínimo 2 nodos y con auto escalado hasta 5 nodos.

- La base de datos se debe cargar el Backup en primera instancia cuando los contenedores inicien.
- La base de datos debe tener una sola replica para evitar problemas de integridad de datos.
- Tiempo respuesta para la aplicación Web no deberá ser mayor a 4 segundos.
- Solo el administrador podrá cambiar los permisos de acceso al sistema.
- Se debe tener un protocolo de comunicación seguro (https), tanto en el servidor como en el cliente.
- La capacidad de usuarios en el sistema puede ser como mínimo 1000 conexiones simultáneas.
- El sistema debe tener la disponibilidad del 98%.
- Promedio de duración de falla no debe ser mayor a 15 min en un nodo.
- Pruebas de rendimiento deben ser ejecutadas con Jmeter[9].

2.4 HERRAMIENTAS

En consideración al problema o los problemas que puedan presentarse durante el desarrollo y lanzamiento de las aplicaciones, además de las diferentes herramientas que ayudan a disminuir estos problemas con respecto al tiempo, a lo económico y que no generen un valor agregado al cliente, se han plateado varias herramientas para llegar a un punto de equilibrio el cual permita automatizar las tareas que son necesarias para generar una aplicación y enviarla a producción. En la implementación se usa dos aplicaciones por el lado del Backend un servidor web, para el Frontend una aplicación con JavaScript.

[9] Jmeter: es una herramienta que fue desarrollada principalmente para realizar pruebas de rendimiento a servidores web, bases de datos, entre otras; es Open Source y está escrita en el lenguaje de programación Java.

2.4.1 DOCKER

A nivel de aplicación tanto para servidor y cliente es recomendable utilizar Docker para construir las imágenes con toda las dependencias necesarias para que funcione en cualquier entorno de contendores, bien sea con Docker o Containerd, la ventaja de usar contenedores es que posteriormente las aplicaciones serán desplegadas en la nube, se pueden testear en un entorno local, además se pueden subir a algún repositorio de imágenes para ser compartidas y asegurar el funcionamiento en otros entornos, con esto se evita problemas de compatibilidad, errores con respecto al Host que los ejecuta, normalmente suelen darse errores con respecto a librerías o versiones de S.O en las que se ejecutan, la aplicaciones estarán aisladas del Host y serán expuestas mediante puertos mapeados entre el contendor y el Host.

2.4.2 DOCKERFILE

La construcción de la imagen estará presidida por los Dockerfile, los que contienen los pasos necesarios para generar la imagen con la aplicación. Por ejemplo, es necesario ejecutar un servidor web con Django, entonces el Dockerfile debe tener una imagen base, la cual deberá ser una con Python, también deberá agregar el proyecto local a la imagen, ejecutar el archivo de librerías necesarias para el proyecto, exponer el puerto entre otros pasos, entonces el Dockerfile tiene la definición de los pasos necesarios para empaquetar la aplicación y que la misma funcione, el resultado deberá ser una imagen luego de la construcción.

2.4.3 DOCKER HUB

Actualmente, existen varios repositorios de imágenes al igual que se usan para versionar código, se pueden versionar las imágenes, ya que cada versión tiene nuevas funcionalidades, por ejemplo, se tiene imágenes con Ubuntu y las diferentes versiones 14.04, 16.04, Python, JDK de Java, Node, generalmente todas las imágenes públicas o de software libre están dentro de Docker Hub, es por

esto por lo que se utiliza para almacenar las imágenes de las 2 aplicaciones con sus diferentes versiones.

2.4.4 IMÁGENES BASE

Para el proyecto se usan varias imágenes que tienen diferentes funciones. Por ejemplo, para compilar la aplicación cliente, la aplicación del servidor, para almacenar los datos, las imágenes base son como una la aplicación por defecto que sirve para ejecutar algún proceso, suponiendo que se necesitara ejecutar un servidor con Wildfly se debe buscar una imagen con Java y posteriormente instalar el servidor o usar una imagen directamente que tenga instalado Wildfly. Las que se utilizan en el proyecto son 4, una imagen con Postgresql que será utilizada para la gestión de la base de datos, otra imagen con el Jdk de java para ejecutar la aplicación con Spring, una más con Node para la aplicación de Angular y finalmente una con el servidor web de Nginx, por cada una de ellas se genera otra que contienen las dependencias necesarias para el funcionamiento.

2.4.5 GITLAB CI

En el capítulo anterior se habla de manera breve de varias herramientas que ayudan a la automatización de los procesos con respecto a la integración y al despliegue continúo de aplicaciones. El proyecto se apega a la integración con Gitlab Ci, simplemente se trata de un archivo yaml que contiene los pasos necesarios para realizar las pruebas, la compilación, la construcción y el despliegue, más adelante se explica a detalle cada proceso. Es una herramienta que agilita todo el proceso para la automatización gracias a los agentes que provee Gitlab que operan sobre el archivo yaml que contiene las instrucciones necesarias.

2.4.6 KUBERNETES

Como orquestador de contenedores se utiliza Kubernetes, es una buena opción para desplegar las aplicaciones, una ventaja importante que tiene con respecto a solamente usar Docker es que permite orquestar a gran escala, escalado horizontal, escalado vertical, fácilmente integración en la nube con cualquier Proveedor, administrar fácilmente los recursos, todo esto se lo define en archivos yaml, no necesita realizar de forma manual porque contiene un CLI que permite ejecutar de manera fácil cualquier comando o archivo de configuración. Los manifiestos contienen las imágenes que se necesiten desplegar, los puertos, volúmenes, variables de entorno, etiquetas y varias más.

CAPÍTULO 3 DISEÑO DE LA ARQUITECTURA

3.1 DEFINICION DE LA ARQUITECTURA

Actualmente, muchas de las empresas aún conservan arquitecturas tradicionales, la idea principal que se tiene con respecto a una arquitectura construida con Kubernetes es optimizar tiempos de entrega, uso de balanceadores de cargas y nodos independientes para las cargas de trabajo.

Se definen dos nodos para el Clúster, dado que en la literatura señala y recomienda usar tres nodos como mínimo para levantar los servicios en producción (Kubernetes, 2020), con el propósito de evitar problema de caídas o rendimiento, dentro del Clúster se definirá la arquitectura para la aplicación que estará en producción, la cual contará con un servicio de base de datos conjuntamente con un volumen para evitar la pérdida de la información en el caso que un Pod termine su ciclo de vida, una aplicación Backend desarrollada con Spring Boot y una aplicación Frontend desarrollada con Angular, la conexión interna entre servidor y base de datos es mediante un servicio de tipo ClusterIp, ya que no será necesario exponer la base de datos al exterior, para la conexión de servidor con el cliente de igual manera se dispone de un servicio de ClusterIp, los dos últimos para poder ser expuestos a los usuarios finales serán mediante un Objeto de tipo Ingress, el cual apunta a los dominios y subdominios y estos a su vez serán expuesto a través de un Balanceador de carga el cual enviará todas las peticiones al Ingress.

También se usa un Objeto ClusterIssuer que se encargara de emitir certificados para que los servicios tengan el protocolo Https, mientras que es necesario disponer de un Dns para que el sitio sea seguro y su respectivo administrador.

3.2 ARQUITECTURA Y COMPONENTES

Analizando la arquitectura de manera general, antes de pasar explicar cada componente y objeto Kubernetes, la arquitectura tiene como único punto de acceso un balanceador de carga el cual apunta dentro del registro Dns al número de subdominios que se vaya a manejar, en este caso como se muestra en la ilustración 12 se tiene dos subdominios. Entonces, se crean dos registros tipo A apuntando a la misma Ip, puesto que el balanceador de carga tomara todas las peticiones y posterior las reenviara al Ingress el cual es la siguiente parte de la arquitectura.

Ingress en este caso actuará como un filtro tomando como referencia el subdominio al cual se envió una petición y reenviando al servicio correspondiente, por ejemplo, si la petición es hacia el subdominio app-docker.tech, el Ingress verifica en primera instancia que ese subdominio este definido y luego verifica que se tenga asociado un servicio por nombre y puerto y reenvía la petición hacia el Deployment correspondiente.

Para este punto existen tres Deployments que se han definido, uno que contiene la base de datos, la aplicación Backend y la aplicación Frontend, el Deployment define el estado deseado de la aplicación, como puede ser el número de réplicas que va a tener la aplicación, por ejemplo, cinco replicas esto para balancear la carga de trabajo, para la base de datos solo se debe tener una réplica para evitar problemas de integridad de datos, además, se debe exponer los puertos que se usan dentro de la aplicación, el 80, 8080 y 5432 son lo que se exponen fuera del contenedor los mismos que están fuertemente ligados con los servicios que actúan como medio de acceso a cada aplicación a través de los selectores, servicios que son de Tipo ClusterIp que son visibles únicamente para el Cluster, estos servicios son los mismos que el Ingress apunta para identificar que aplicación debe responder cuando llega una nueva petición por el balanceador de carga.

Cabe mencionar que cada parte de la arquitectura se la definió mediante manifiestos declarativos de Kubernetes, puesto que es más fácil actualizar una aplicación o la base de datos con tan solo aplicar los nuevos cambios dentro de los archivos y no eliminando toda la aplicación y así evitar el despliegue desde cero. Por otro lado, el fin que se tiene para la aplicación como producto de Software que soportará la arquitectura, la misma que está definida en el objetivo específico número tres, antes de pensar en una aplicación independiente a las tecnologías que se hayan usado para el desarrollo, el objetivo se centra en el desarrollo de una aplicación basada en microservicios, entonces, es por ello que dicho objetivo está enfocado en dar una solución de microservicios para una aplicación sin importar la tecnología en la que se desarrolló. Aunque más adelante se mencionan de manera constante las tecnologías usadas en el desarrollo esto es más que nada porque dentro de los manifiestos se deben especificar ciertas condiciones que cada tecnología hace uso, condiciones simples como puertos, variables de entorno o imágenes base.

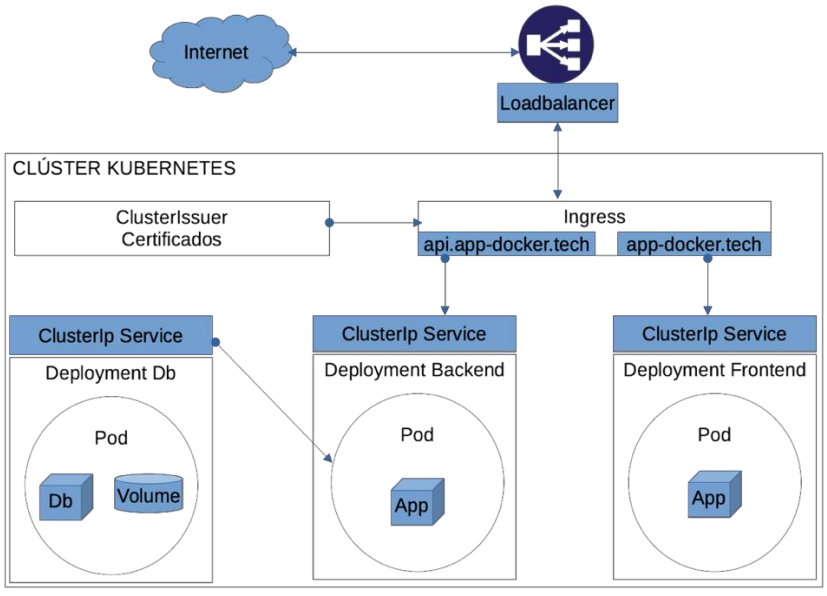


Ilustración 12 Arquitectura principal

Cada componente tiene un papel muy importante dentro de la arquitectura ya que están fuertemente relacionadas para el funcionamiento correcto de la aplicación que está expuesta al usuario final.

Componentes:

- Nodos: los nodos son los Droplets que forman el Clúster de Kubernetes, uno Máster y otro Worker, los cuales equilibran la carga.
- Deployment base de datos: contiene al estado deseado de la base de datos, por ejemplo, la información que migrara al momento de iniciar el Pod.
- Volume: es el espacio que puede estar dentro del Clúster o fuera del mismo, la idea principal es que la información de la base de datos sea íntegra y no llegue a un punto de pérdida de información.
- Deployment Backend y Frontend: contiene la aplicación con Spring y Angular que previamente fue construida con Docker.
- Servicio ClusterIp: expone el servicio de la base de datos, Backend y Frontend de manera visible únicamente para el Clúster.
- Balanceador de carga: es una máquina a nivel de proveedor el cual recibe las peticiones y las envía al Ingress, escucha por los puertos 443 y 80.
- Ingress: contiene los dominios y subdominios que apuntan a cada servicio de tipo ClusterIp especificando el puerto.
- ClusterIssuer: objeto que emite los certificados Ssl tanto para el servidor y el cliente.

3.3 DEFINICIÓN MANIFIESTOS

Cada Deployment de la aplicación cumple una función bien definida, por lo que se desacoplo en varias partes, para mantener en un futuro cambio de versiones en cuanto a manifiestos de Kubernetes o simplemente la sintaxis.

3.3.1 BASE DE DATOS MANIFIESTO

Se Especifica el estado deseado de la base de datos, con un único Pod siempre activo a pesar de que son efímeros, el Nodo Máster siempre va a levantar uno, de igual manera la información previa con la que se genera la base de datos, por ejemplo, tablas y datos necesarios.

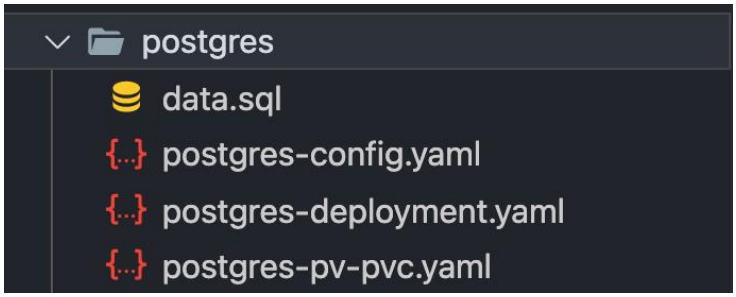


Ilustración 13 Manifiestos para la base

Contiene la información inicial para restaurar la base de datos por primera vez cuando se levanta el Deployment, además, se define la configuración del volumen donde se almacenará la información de Postgresql, para evitar la pérdida de información, también, contiene las credenciales de acceso a la base de datos.

Tabla 4 Configmap Postgresql

```
apiVersion: v1
kind: ConfigMap
metadata:
name: postgres-config
data:
initdb.sql: |
```

El ConfigMap contiene la estructura de la base de datos y la información de todas las tablas, se ejecuta solo la primera vez que levanta la base de datos.

Tabla 5 Deployment base de datos

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-deployment
  labels:
    app: ups
    db: postgres
spec:
  selector:
    matchLabels:
    app: ups
      db: postgres
  replicas: 1
  template:
  metadata:
     labels:
       app: ups
       db: postgres
   spec:
   containers:
       - name: postgres
         image: postgres:10
         env:
           - name: POSTGRES_USER
             value: postgres
           - name: POSTGRES_PASSWORD
```

```
          value: ups2021
        ports:
          - containerPort: 5432
        volumeMounts:
          - name: postgres
            mountPath: /var/lib/postgresql/
          - name: postgres-initdb
            mountPath: /docker-entrypoint-initdb.d
      volumes:
        - name: postgres
          persistentVolumeClaim:
          claimName: postgres-pvc
        - name: postgres-initdb
          configMap:
            name: postgres-config
```

Para el Deployment de la base de datos se debe especificar configuraciones extra, en primer lugar, se define un apiVersion que es la versión dependiendo el objeto de Kubernetes que se desea levantar, se utiliza la versión “apps/v1” para los objetos de tipo Deployment, las etiquetas son opcionales, su principal uso es para hacer búsquedas en el caso que se tengan varios objetos desplegados, con las etiquetas será más fácil obtener Pods, Services, entre varios más.

Los selectores son para identificar uno o varios objetos de cualquier tipo en base a etiquetas que tengan coincidencia. La opción réplicas, hace referencia al número de Pods que se van a ejecutar y que contiene la base de datos para este caso. La parte de spec, son todas las especificaciones de los contenedores, como el nombre que tiene el Pod, la imagen base que se tiene, en este caso

Postgresql con la versión 10, también las credenciales de conexión, el puerto del contendor que es el 5432 para Postgresql.

Finalmente, la configuración de los volúmenes, postgres-config que está encargado de migrar toda la información de las tablas con sus respectivos datos y postgres-pvc que es el volumen en el que se persistirá la información en caso de que el Clúster elimine el Pod.

Tabla 6 Servicio base de datos

```
apiVersion: v1
kind: Service
metadata:
  name: postgres-service
  labels:
    app: ups
    db: postgres
spec:
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 5432
      targetPort: 5432
  selector:
    app: ups
    db: postgres
```

Para acceder a la base de datos no es suficiente con el Deployment, se puede acceder sin necesidad de servicio solamente usando la Ip del Pod, pero como se mencionó el Pod es efímero y puede morir en cualquier momento y su Ip va a cambiar, por ello se necesita exponer de manera general al Deployment, por este motivo se utiliza el servicio de tipo ClusterIp especificando el puerto que se exponen y usando el selector para hacer Match con respecto al Deployment. Entonces, el apiVersion para objetos de tipo Service es v1, el tipo de servicio es ClusterIp, se accede al servicio a través del puerto 5432, recordando que este servicio será únicamente visible dentro del Clúster.

3.4.1 BACKEND MANIFIESTO

Para la parte del servidor se hace uso de una imagen con Java puesto que la aplicación está desarrollada con el Framework Spring Boot, se inicia de una imagen previamente construida (a mano solo la primera vez), ya que posteriormente la construirá de manera automática usando los Pipelines de Gitlab.

Tabla 7 Deployment Backend

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
  labels:
    app: ups
    spring: backend
spec:
  selector:
    matchLabels:
    app: ups
      spring: backend
  replicas: 1
  template:
  metadata:
      labels:
        app: ups
        spring: backend
    spec:
      containers:
```

```
      - name: backend
        image: fcusco96/spring:v1
        ports:
          - containerPort: 8080
```

El Deployment es muy similar a la base de datos con la diferencia que se pueden generar varias réplicas, tampoco se utilizan volúmenes ni variables de entorno, simplemente se especifica el puerto de conexión, como se trata de una aplicación Java con Spring, se tiene un servidor embebido de aplicaciones, el cual es Tomcat y su puerto por defecto es el 8080, por ese motivo se expone dicho puerto dentro del contenedor, también se especifica el número de réplicas con 2, con esto se logra la distribución de carga, cada vez que haya nuevas peticiones al servidor.

Tabla 8 Servicio Backend

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
  labels:
    app: ups
    spring: backend
spec:
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
  selector:
    app: ups
    spring: backend
```

Los más importante que se define es el puerto que es el mismo que está expuesto en el Deployment, ya que simplemente se mapeará al puerto de Container y redirigirá todo el tráfico hacia los Pods.

3.4.2 FRONTEND MANIFIESTO

Para la parte de cliente es necesario configuraciones extra ya que está desarrollada con Angular y se debe compilar como un proyecto solo con JavaScript y en modo de producción, pero todas las configuraciones necesarias están demostradas en el siguiente capítulo.

Tabla 9 Deployment Frontend

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: angular-deployment
  labels:
    app: ups
    angular: frontend
spec:
  selector:
    matchLabels:
    app: ups
      angular: frontend
  replicas: 1
  template:
    metadata:
    labels:
        app: ups
        angular: frontend
    spec:
      containers:
        - name: frontend
```

```
image: fcusco96/angular:v1
ports:
  - containerPort: 80
```

Conociendo que toda aplicación desarrollada en Angular su puerto por defecto es el 4200, el estado deseado expone el puerto 80, para resolver este problema se usa un servidor web Nginx el cual estará encargado de enviar las solicitudes a la aplicación con Angular.

El servicio para el Deployment simplemente apunta al puerto 80, no es necesario configuraciones extras, si todo está bien configurado el Clúster deberá crear todos los manifiestos según las especificaciones, para verificar que todo este correcto se debe ejecutar el comando de la figura 14. Todos los manifiestos tienen una etiqueta en común que es “app=ups”, permite traer la información que tenga esa etiqueta, si los manifiestos no tuviesen esta etiqueta el comando no mostraría ningún resultado o peor aún si existieran más aplicaciones en ejecución traería todo sin importar la etiqueta, por tal motivo es importante etiquetar a las aplicaciones dentro de una clase o tipo.

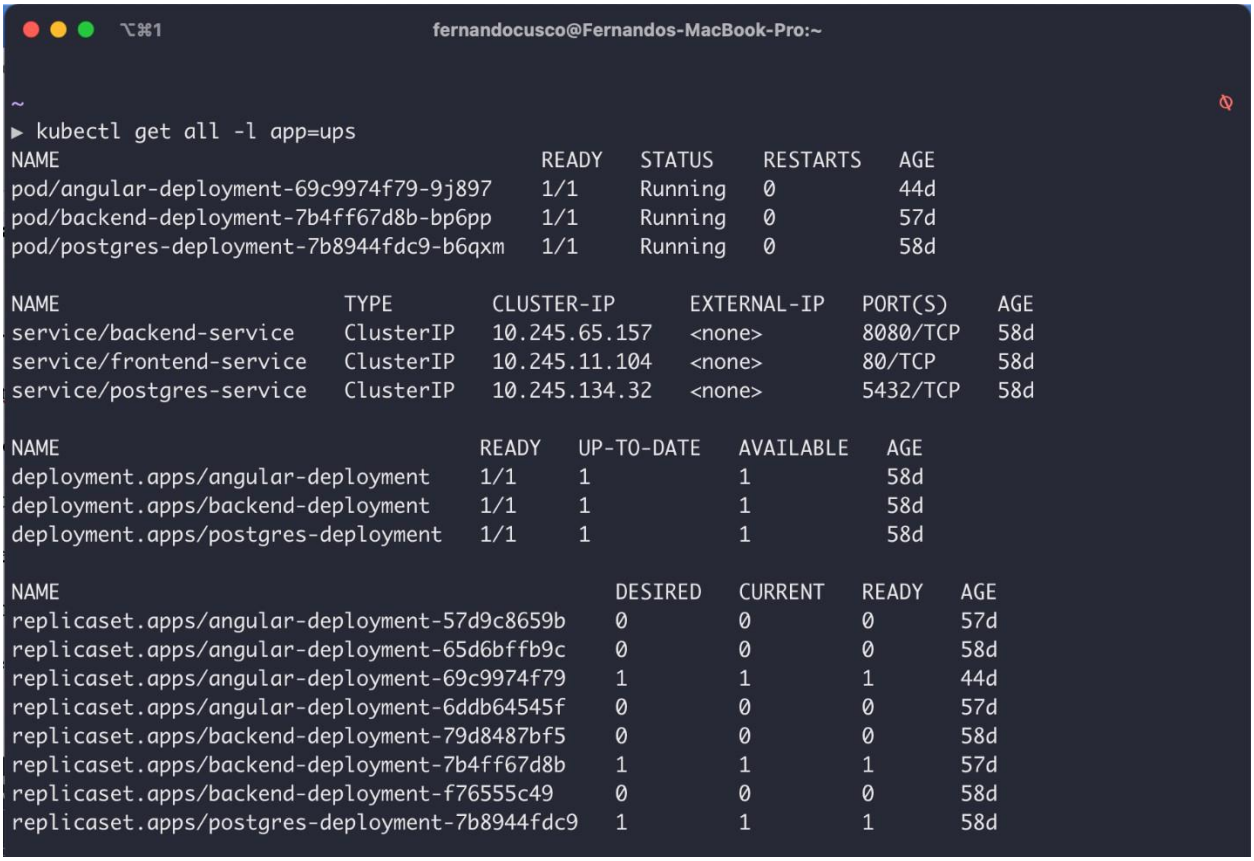


Ilustración 14 Información general

Como se muestra en la ilustración 14 se puede observar todo lo que se creó anteriormente, desde la base de datos, el servidor con Spring y el cliente con Angular, pero además muestra información extra que son los ReplicaSet que están fuertemente relacionados con los Pods y el número de réplicas para cada uno.

3.5 PIPELINE

Pipeline traducido al español significa “tubería” y eso justamente lo que se intenta desarrollar dentro de un ambiente con DevOps, básicamente se canalizan varios Steps (pasos) para llegar a un resultado final, puede existir excepciones durante la ejecución de algún paso el cual puede terminar con el proceso o se puede controlar de alguna manera para ejecutar otro paso, pero con todo esto existen varias herramientas como se muestra en el capítulo 1, cada uno se las define de manera

diferente dependiendo del lenguaje de programación y el propósito para las cuales fueron desarrolladas.

Analizando al Pipeline actual, los pasos definidos son:

- Test
- Build
- Deployment

En el paso de Test se ejecutarán principalmente las Pruebas definidos dentro del proyecto de Angular, precisados con la librería Jest, que es un Framework para ejecutar Test, al momento de validar las Pruebas y en el caso de generar algún error el proceso se detiene y no ejecutara los pasos siguientes. En el paso de Build construirá la imagen que contiene el código generado para producción con la imagen base de Nginx, que simplemente subirá a DockerHub con un Tag único, el mismo que usará la variable de entorno "GILTLA-COMMIT-SHA", que provee el id único por cada Commit de esta manera no existirá problema al subir la imagen.

Finalmente, el Deployment, proceso que actualizará el contenedor con la nueva imagen y como anteriormente ya estaba definido el Deployment y expuesto su servicio, evitará que este se detenga y existan problemas con los usuarios finales.

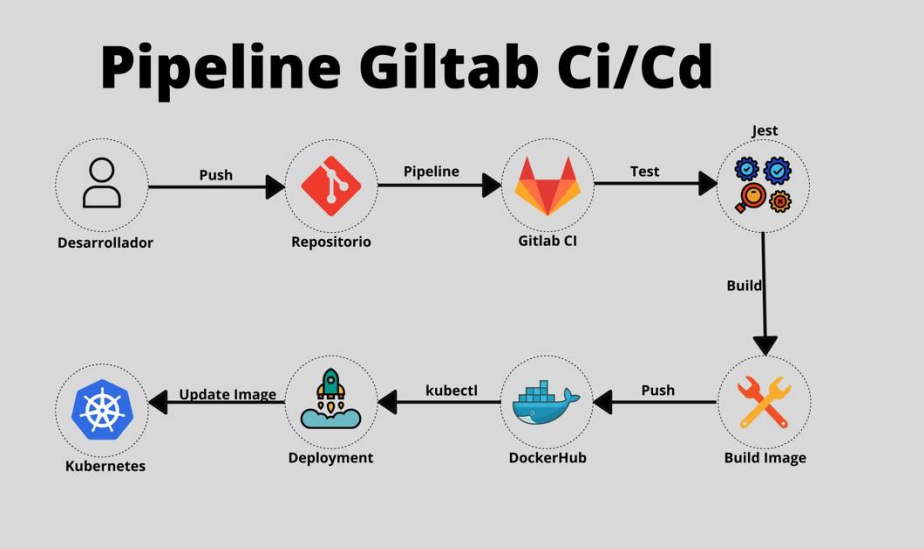


Ilustración 15 Pipeline Gitlab

CAPÍTULO 4 IMPLEMENTACIÓN DE LA ARQUITECTURA PARA REALIZAR INTEGRACIÓN Y DESPLIEGUE CONTINUOS.

4.1 PROVEEDORES CLÚSTER KUBERNETES EN LA NUBE

Para el despliegue en la nube el Clúster de Kubernetes se seleccionó de entre 4 proveedores, los cuales son Google Cloud Computing, Amazon Web Service, Digital Ocean y Azure, a continuación, se muestra una tabla de los diferentes precios y características de hardware.

Tabla 10 Costos proveedores Cloud

CLÚSTER KUBERNETES					
Proveedor	Memoria	CPU	Nodos	Propósito	Precio
Google Cloud	3.75	1	1	General	$24.27/mes
	16	4	1	Optimizado	$121.93/mes
	1433	96	1	Memoria Optimizada	$5443.25/mes
DigitalOcean	1	1	1	Desarrollo	$10/mes
	2.5	2	1	Optimizado	$40/mes
	6	2	1	General	$60/mes
	13	2	1	Memoria Optimizada	$80/mes
Azure	2	2	1	Básica	$36.21/mes
	8	2	1	Optimizado	$85.41/mes
AWS			1	General	$73/mes

Cada proveedor tiene precios y propósitos diferentes, pero existen diferencias que son más importantes con respecto a la administración del Clúster, por ejemplo, Google Cloud por el hecho de haber desarrollado esta tecnología tiene una integración más adecuada para trabajar con respecto a los usuarios, la administración de servicios, la administración de registros Dns, permisos con respecto al Clúster, un usuario que no esté dentro del proyecto pero tenga las credenciales del Clúster para una administración por línea de comandos simplemente no podrá ejecutar ningún comando, esto se debe a la fuerte integración de usuario de Clúster y usuario de Google.

A diferencia de usar el servicio de Digital Ocean, cualquier usuario que tenga las credenciales del Clúster puede ejecutar la línea de comandos. Además, Google Cloud provee volúmenes para la integración con Kubernetes, esta es una gran ventaja que se tiene ya que no es necesario apuntar a servicio de terceros para asegurar la información.

4.2 IMPLEMENTACION KUBERNETES, DNS, GITLAB Y TERRAFORM

4.2.1 CONFIGURACIÓN DEL DOMINIO DNS

Se utilizó un dominio con varios subdominios que se son para el ingreso a las aplicaciones que se despliegan dentro de Clúster, el proveedor de dominio es Hostinger, como todo será controlado desde DigitalOcean, se configura los Nameservers de Hostinger apuntando a los Nameservers que provee DigitalOcean.

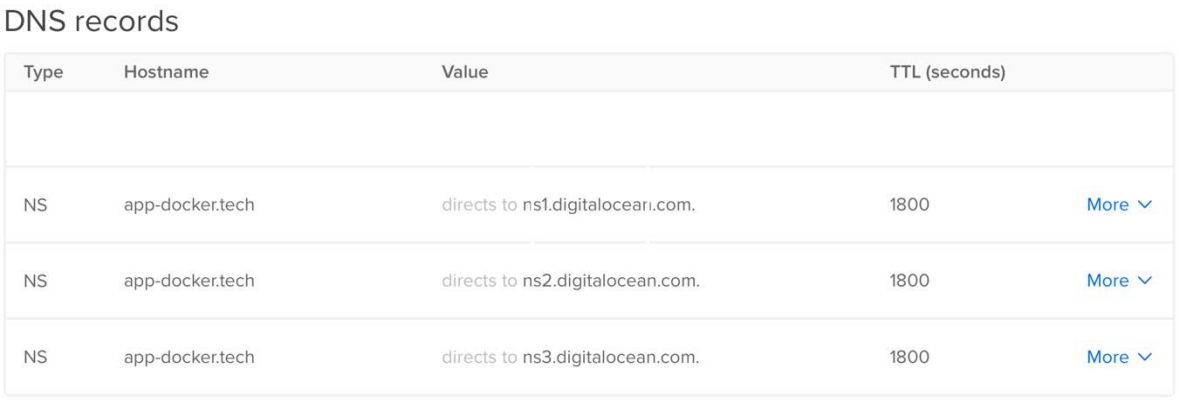


Ilustración 16 *Nameserver DigitalOcean*

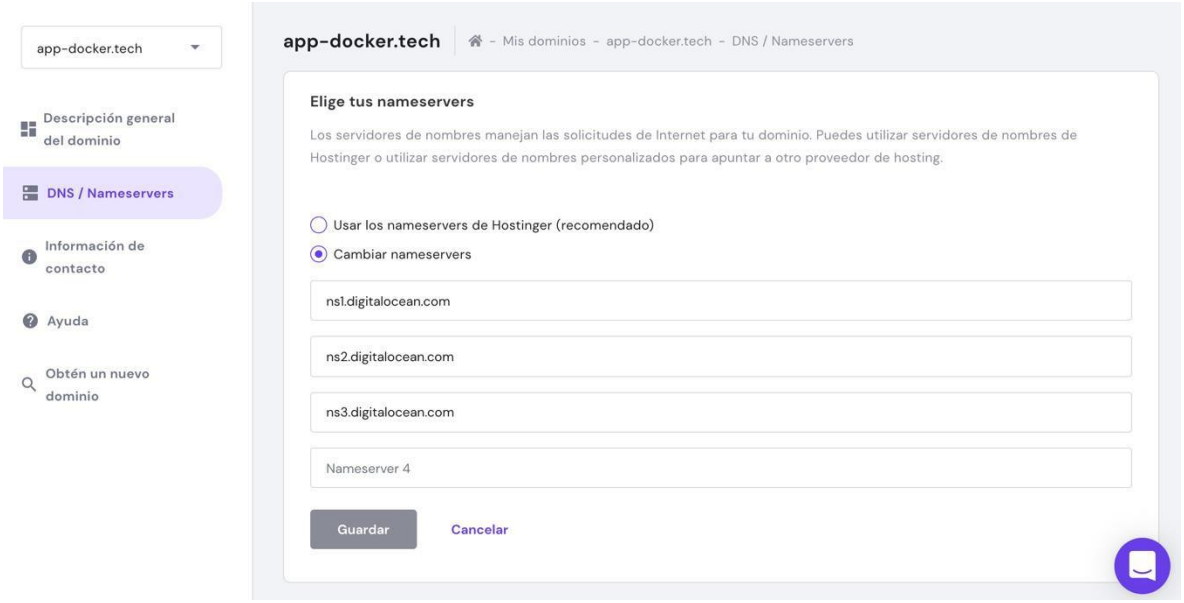


Ilustración 17 Configuración Nameservers hostinger

Se cambia los Nameservers a los de Digital Ocean, con esta configuración todos los registros tipo A, AAAA, CNAME, etc, pasan a ser administrados por Digital Ocean.

4.2.2 IMPLEMENTACIÓN TERRAFORM

Se define una arquitectura básica para un Clúster de Kubernetes a partir de código usando la herramienta de Terraform, además, soporta muchos proveedores de infraestructura en la nube como Amazon Web Service, Google Cloud, DigitalOcean, entre otros. A continuación, se

demuestra cómo se levanta en DigitalOcean, para empezar, se debe solicitar un Token de acceso al proveedor.

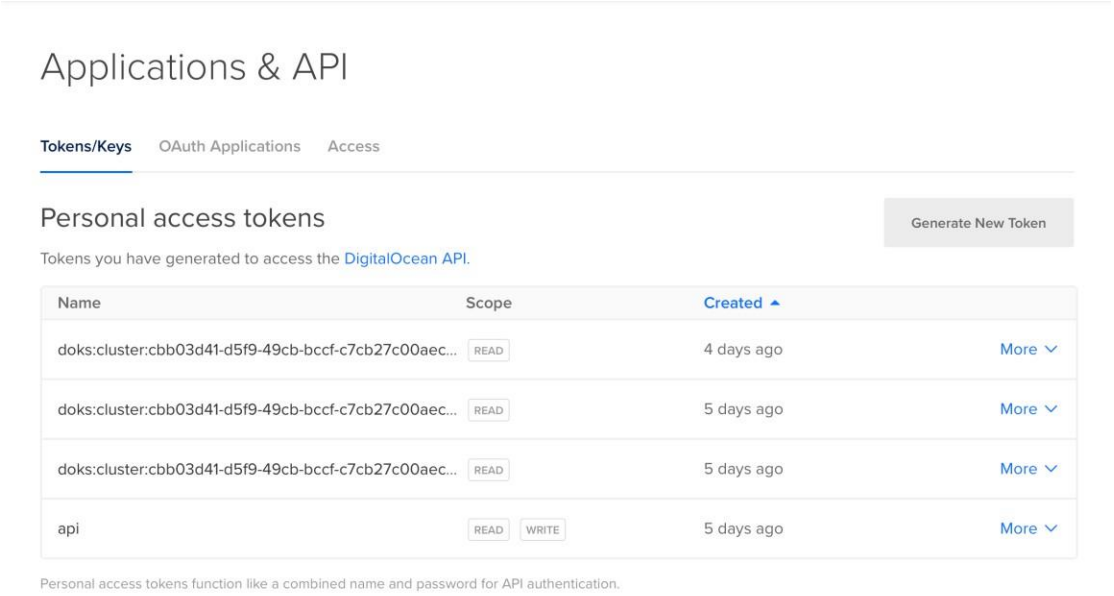


Ilustración 18 Token DigitalOcean

Mediante código se define el Clúster de Kubernetes, la administración de Dns, el Token para la creación de recursos, las claves Shh para acceder a los Droplets que posteriormente se levantan, también se pueden levantar varios Droplets que contengan aplicaciones ya instaladas, por ejemplo, maquinas con Docker y con sus contenedores respectivos.

```
provider "digitalocean" {
    token = "d8bd6f24f25918e6513bc7cf0ce294682ada5e81c2********************"
}
resource "digitalocean_domain" "base" {
  name = "app-docker.tech"
}
```

Ilustración 19 Token DigitalOcean

Se configura el proveedor usando el Token generado previamente dentro de Digital Ocean todo esto para levantar de manera automática de las máquinas necesarias para el Clúster y las

configuraciones extra como se muestra en la ilustración 19 para administrar los registros Dns de un dominio.

```
resource "digitalocean_kubernetes_cluster" "cluster" {
    name = "cluster"
    region = "nyc1"
    version = "1.19.3-do.3"

    node_pool {
        name = "workers"
        size = "s-2vcpu-4gb"
        node_count = 3
    }
}
```

Ilustración 20 Clúster Kubernetes usando Terraform (Terraform, 2020)

Se configura un recurso de tipo Clúster Kubernetes, en la que se especifica el nombre que va a tener el Clúster, la región o zona donde se levantará toda la instancia o el grupo de nodos, la versión de Kubernetes y Pool de nodos, con ciertas características como el nombre, la capacidad de Hardware y el número físico de nodos.

```
resource "digitalocean_kubernetes_node_pool" "kubernetes_autoescaling" {
  cluster_id = digitalocean_kubernetes_cluster.foo.id
  name       = "kubernetes_autoescaling"
  size       = "s-2vcpu-4gb"
  auto_scale = true
  min_nodes  = 2
  max_nodes  = 7
}
```

Ilustración 21 Escalado automático (Terraform, 2020)

Para la parte de auto escalado de nodos se puede configurar dentro del mismo proveedor o simplemente usar Terraform para levantar el Clúster con auto escalado automático, de esta manera hacerlo todo a partir de código y tener un control central desde un simple archivo de configuración.

Cabe mencionar que la sintaxis puede variar dependiendo del proveedor que se esté utilizando, pero en términos generales la sintaxis es similar.

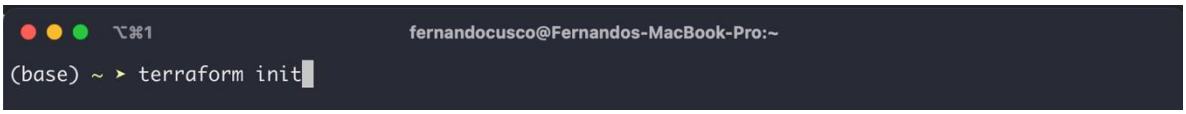


Ilustración 22 Inicializar proyecto

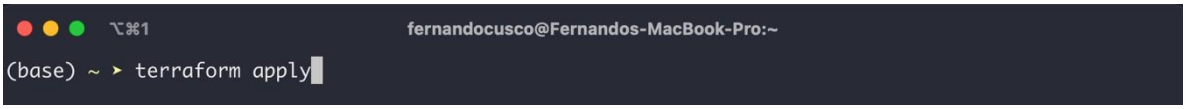


Ilustración 23 Generar la infraestructura en la nube

Con toda la configuración lista se aplican los cambios y de manera automática se genera la infraestructura con todo lo especificado, el tiempo despliegue dependerá del tipo de recurso que se configuró, si llegara a ocurrir un error de sintaxis u otro tipo de error, la consola mostrará el error específico.

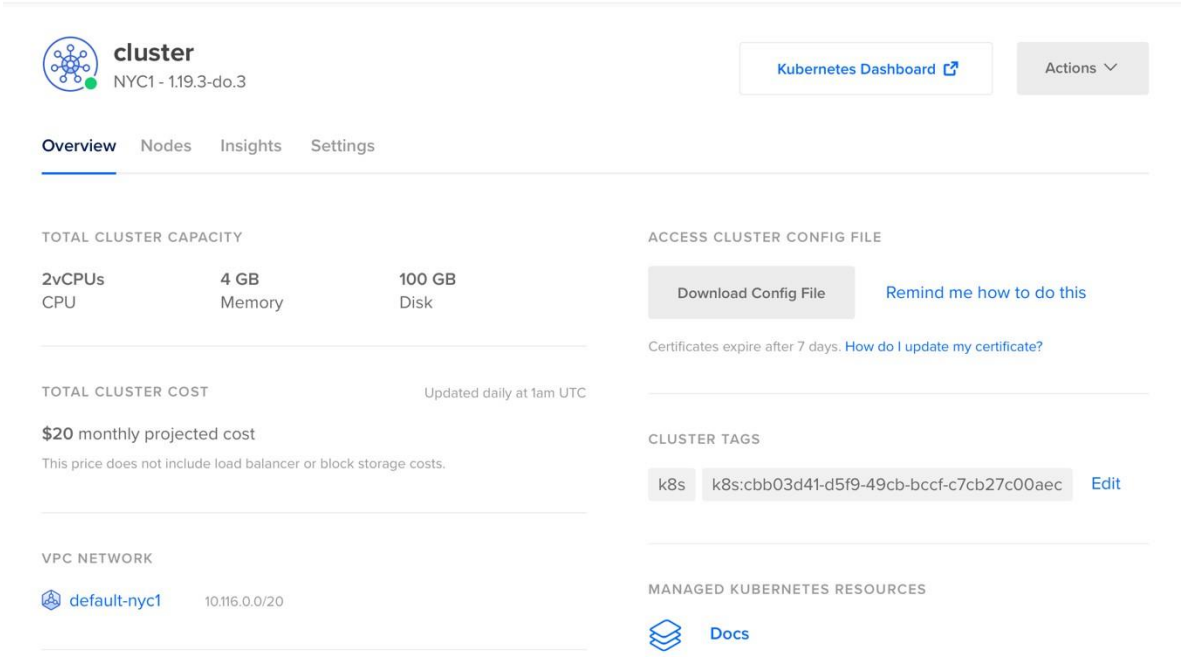


Ilustración 24 Clúster Kubernetes

4.2.3 CONFIGURACIÓN PARA LA INTEGRACIÓN Y DESPLIEGUE CONTINÚO USANDO GITLAB CI

4.2.3.1 CONFIGURACIÓN PROYECTO SPRING

Para la presente arquitectura, se han definido varias herramientas, para desplegar las aplicaciones a producción desde la parte de integración hasta el despliegue continúo.

Para la parte del Backend, se utiliza el Framework Spring Boot de Java, aunque la configuración es casi similar si se tratara de aplicar a otros lenguajes de programación o Frameworks, por lo general Docker tiene una imagen para manejar cada lenguaje o Framework con todo lo necesario para ejecutar una aplicación dependiendo del lenguaje. Continuando, el proyecto con Spring Boot contiene la lógica de negocio, modelos mapeados a la base de datos, seguridad con Spring Security y además expone los servicios Rest para ser consumidos en la aplicación desarrollada en Angular, para la implementación de CI/CD se debe tener en cuenta dos puntos muy importantes, por un lado la configuración correcta del Dockerfile, archivo que contiene los pasos necesarios para generar una imagen propia y que garantiza el correcto funcionamiento de la aplicación en cualquier entorno de contenedores y de igual manera en Kubernetes. Por otro lado, la configuración del manifiesto de Gitlab que permite la construcción y el despliegue de la aplicación a partir de un simple Commit al repositorio, desde este punto el usuario desarrollador se desvincula de lo que esté en la configuración, ya que todo será de manera continua.

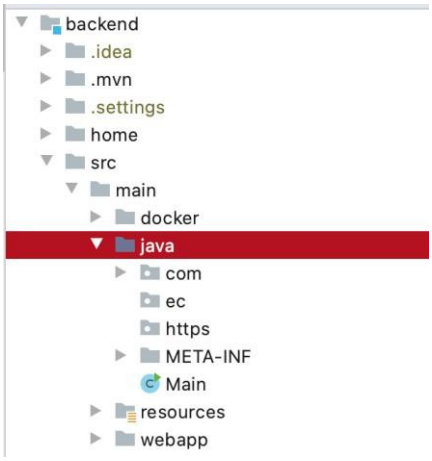


Ilustración 25 Estructura proyecto Spring

Tabla 11 Dockerfile Spring

```
FROM java:openjdk-8-jdk-alpine
COPY certificado1.der certificado1.der
COPY certificado2.der certificado2.der
COPY certificado3.der certificado3.der
COPY certificado4.der certificado4.der
ADD /target/file.jar //
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-Dspring.profiles.active=container","-jar","/file.jar"]
```

La imagen final tiene varias configuraciones para que funcione de manera correcta la aplicación con Spring, las configuraciones aplicadas son las siguientes:

- FROM java:openjdk-8-jdk-alpine: la imagen base con la que se construirá la imagen propia es una que tiene Java en la versión 8.
- COPY files.der: se copia desde el proyecto local todos los archivos de las claves públicas a la imagen.
- RUN: permite ejecutar comandos (Linux) dentro del contenedor al momento de construir la imagen, para el caso, se agregarán los certificados al baúl de certificados confiables de Java.

- ADD /target/fle.jar //: se agrega el archivo .jar a la imagen, el .jar debe ser generada a mano o automáticamente.
- ENTRYPOINT: para finalizar lo que se hace es ejecutar el archivo .jar, este comando se ejecutara únicamente cuando la imagen generada sea puesta en ejecución.

Como repositorio de imagen se usa DockerHub para almacenar la misma y controlar las versiones de cada cambio que se realice dentro del proyecto, con esto se asegura que el proyecto funcione en cualquier entorno Cloud.

Ilustración 26 Build imagen

La construcción de la imagen va a depender de cuantos pasos se hayan definido en el Dockerfile, además se verifica que la imagen base esté disponible de manera local, caso contrario lo que hace Docker es buscar directamente en su repositorio DockerHub e intentará descargarla, puede haber excepciones, ya que las imágenes pueden estar de manera privada y el usuario con el que se inicia sesión debe tener los permisos para descargar, pero generalmente las imágenes base son públicas, otra excepción son los errores típicos de sintaxis que pueden ser ocasionados por las diferentes versiones que maneja Docker para sus archivos Dockerfiles o simplemente errores.

Cuando se trata de generar imágenes propias es importante usar el nombre de usuario de DockerHub seguido de una barra(/) luego con un nombre personalizado y terminando por especificar el Tag usando los dos puntos(:) para la versión de la imagen, ya que Docker no permite poner un nombre simple, por ejemplo, generar una imagen con el nombre “mysql”, esto con total seguridad va a dar una excepción, ya que este tipo de nombres son usados para imágenes oficiales,

por ello es importante anteponer el nombre de usuario, si todo ha salido bien en la construcción se debe tener algo similar a lo que se muestra en la siguiente ilustración.

Ilustración 27 Imágenes

Es muy importante verificar la imagen, principalmente que contenga todo lo que se ha agregado y configurado previo a la construcción, por este motivo se recomienda ejecutar dentro de un contendor antes de subir a DockerHub u otro repositorio de imágenes, simplemente se debe ejecutar el contenedor y como se especificó en el ENTRYPOINT que debe ejecutar el .jar, la ejecución del contenedor debe mostrar algo similar a la siguiente ilustración, que sólo iniciar el contenedor ver la ejecución del .jar. Este caso aplica, ya que se especificó en los ENTRYPOINT ejecutar el archivo jar al iniciar el contendor.

Ilustración 28 Ejecución contendor Docker

La imagen únicamente por una sola vez se construye y se la sube manualmente, ya que esto lo realizará el Pipeline de Gitlab de manera automática y transparente al usuario, finalmente dentro del repositorio de imágenes estará la imagen subida.

Ilustración 29 Push imagen

4.2.3.2 CONFIGURACIÓN PIPELINES SPRING

Se definen varios pasos para realizar la canalización de la construcción y Deployment de la aplicación a partir del Dockerfile. Como primer paso dentro del manifiesto se define la construcción de la imagen basándose en el al archivo Dockerfile, para subir la imagen al repositorio de DockerHub se hace uso de variables de entorno que se definen dentro del proyecto en Gitlab, en este caso se define el usuario y la contraseña de DockerHub para iniciar sesión de manera automática y subir la imagen, todo esto se ejecuta dentro de Clúster de Kubernetes, con la ayuda del Agente que provee Gitlab, más adelante se explica cómo configurar el Clúster dentro del proyecto de Gitlab. El paso de la construcción se ejecuta tanto en la rama máster y en la rama desarrollo.

Tabla 12 Build Backend

```
docker-build-master:
  image:  docker:latest
  stage: build
  services:
    - docker:18.09.7-dind
  before_script:
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" docker.io
  script:
    - docker run -t --rm -v "$(pwd)":/usr/src/mymaven -w /usr/src/mymaven maven:3.3-jdk-8 mvn clean install
    - docker build --pull -t fcusco96/spring:"$CI_COMMIT_SHA" .
    - docker push fcusco96/spring:"$CI_COMMIT_SHA"
  variables:
    DOCKER_HOST: tcp://docker:2375
    DOCKER_TLS_CERTDIR: ""
```

```
only:
  - master
```

El primer paso definido es el “build” de la imagen, todo lo que se hizo de manera manual ahora se ejecutará automáticamente siempre que haya un cambio en la rama master del repositorio, ya sea por un Commit directo o un Merge, lo que se desarrolló principalmente es, usar la imagen de Docker, siempre utilizará la última que esté disponible, en el paso previo se indica hacer un login hacia “docker.io” el sitio oficial, ya que las imágenes y sus distintas versiones se alojan en el mismo, por lo que se ejecuta el comando “docker login” especificando las credenciales que están como variables de entorno dentro del proyecto

Es importante agregar todos estos comandos previos, ya que la consola para CI/CD no admite ingresar nada manualmente, posteriormente se ejecutan los mismos comandos que fueron utilizados de manera manual, con uno adicional, el primero y el más importante trata de generar el archivo .jar que posteriormente se empaquetará dentro de la imagen, se especifica un volumen el cual esta enlazado con el proyecto, de esta manera tomará toda la carpeta para generar el .jar, se especifica el WORKDIR que es similar al “cd” de Linux, se ubica en la ruta donde esta maven dentro del contenedor para crear el .jar y su salida será dentro del repositorio de Gitlab en la carpeta target la misma que se usa en el Dockerfile.

A continuación, se construye la imagen usando como Tag la variable de entorno que almacena el Commit Sha, de esta manera siempre que haya un cambio se asegura que se suba la última versión de la aplicación con los nuevos cambios y finalmente se sube esa imagen versionada al repositorio en el caso que las credenciales sean correctas.

Tabla 13 Stage Deploy

```
deploy:
  stage: deploy
  image:
    name: bitnami/kubectl
    entrypoint: [""]
  script:
    - kubectl -n default set image deployment/backend-deployment
backend=fcusco96/spring:"$CI_COMMIT_SHA"
```

Para la parte del Deployment es algo más sencillo, se indica la imagen en este caso se utiliza una para ejecutar comandos de Kubernetes, ya que lo hará desde Gitlab y no ingresando directamente al Clúster, conociendo el nombre de la imagen y su respectiva versión, simplemente se hace un update del Deployment que previamente se ejecutó con el estado deseado de la aplicación, no es necesario detener el servicio, ya que Kubernetes tiene la capacidad de hacer este trabajo en cuestión de segundos, eliminara todos los Pods asociados con el Deployment y creará una nueva instancia con la nueva versión.

El script define el estado deseado del Deployment, de esta manera se controlan los recursos necesarios que debe usar ese Pod, el número de réplicas, los puertos.

4.2.3.4 CONFIGURACIÓN PROYECTO ANGULAR

Lo que se necesita antes de nada es la configuración para la conexión con respecto al servidor el cual expone los servicios Rest, la configuración del Dockerfile, la configuración del Servidor Web con Nginx y finalmente la configuración del manifiesto de Gitlab para CI/CD.

A partir del proyecto con Angular se tiene que generar el proyecto en modo de producción que básicamente se compone de archivos JavaScript, Html y Css, con esto no se depende de angular para levantar el proyecto, sino que usará Nginx para alojar al proyecto y que el mismo reenvíe las peticiones, la ventaja que se tiene al usar este servidor Web es que permite mejorar el rendimiento en cuanto a conexiones, otra ventaja muy importante es el almacenamiento de contendido estático

en caché por periodos cortos de tiempo, alcanzando hasta 1000 archivos en caché, teniendo en cuenta que el proyecto se genera en modo producción entonces se crean varios archivos estáticos, también está disponible la compresión gzip, ayudando a disminuir el tamaño de entrega de archivos.

Tabla 14 Dockerfile Angular

```
FROM node:10.16.3 as build-stage
WORKDIR /app
COPY package*.json /app/
RUN npm install
COPY ./ /app/
ARG configuration=production
RUN node patch.js
RUN rm /app/src/app/test/operacion.test.ts
RUN npm run build -- --output-path=./dist/out --configuration $configuration

FROM nginx:1.15
COPY --from=build-stage /app/dist/out/ /usr/share/nginx/html
COPY ./nginx-custom.conf /etc/nginx/conf.d/default.conf
```

El Dockerfile contiene 2 configuraciones con varios pasos cada uno, primero genera el proyecto en modo producción:

- FROM node:10.16.3 as build: el Proyecto está desarrollado con una versión especifica de Node y Angular, por ello se especifica dicha versión en la imagen base.
- WORKDIR /app: el lugar de trabajo será a partir de la ruta especificada dentro del Container.
- COPY package*.json /app/: se necesita copiar los archivos que contienen las dependencias que se usan en el proyecto, para posteriormente instalarlas.
- RUN npm install -g @angular/cli@6.0.8: se instala el cli de angular en una versión específica.
- RUN node patch.js: se ejecuta un parche para la encriptación (propio del proyecto).

- RUN npm run build -- --output-path: se generar el Proyecto en modo producción y se guarda la salida en la carpeta dist.

La segunda configuración contiene:

- FROM nginx:1.15: la imagen base será del Servidor Web para alojar la aplicación en modo producción.
- COPY –from=build /app/dist/out /usr/share/nginx/html: se copia la carpeta generada en el paso anterior.
- COPY ./nginx-custom.conf /etc/nginx/conf.d/default.conf: se reemplaza la configuración del servidor con una personalizada.

Ilustración 30 Build imagen

Ilustración 31 Push imagen

4.2.3.5 CONFIGURACIÓN PIPELINES ANGULAR

La configuración es similar a la que se utiliza para la aplicación de Spring, con la diferencia que no solo se define la parte de construcción, también es la encargada de generar la imagen usando el archivo Dockerfile y subirla al repositorio de Imágenes de DockerHub y la del Deployment que simplemente actualiza la imagen con la nueva versión, además contiene un paso extra para la parte de los Test de la aplicación, al momento de ejecutar el Commit lo primero que se ejecutará es el Test, simplemente ejecuta los Test definidos usando la librería Jest de Angular.

Tabla 15 Gitlab Ci Test

```
test:
  stage: test
  image: node:10.16.3
  before_script:
    - npm install
  script:
    - npm test
  only:
    - master
```

El Pipeline luego de ejecución deberá mostrar todos los pasos ejecutados, si el proceso terminó con éxito o se encontró alguna excepción y se detuvo el proceso.

Ilustración 32 Pipeline ejecutado

En este caso todos los pasos se ejecutaron, el primero hace referencia a las Pruebas, el segundo construye la imagen y el tercero a la actualización del Container con la nueva imagen y la aplicación deberá mostrar los cambios de manera casi inmediata.

4.2.3.5 INTEGRACIÓN GITLAB Y CLÚSTER KUBERNETES

La canalización de los Pipelines se configura en Gitlab, el repositorio de código del proyecto debe tener lo necesario para poder conectarse y ejecutar los pasos de CI y CD, entonces lo primero que se debe hacer es configurar las variables de entorno que Gitlab provee o crear las variables propias.

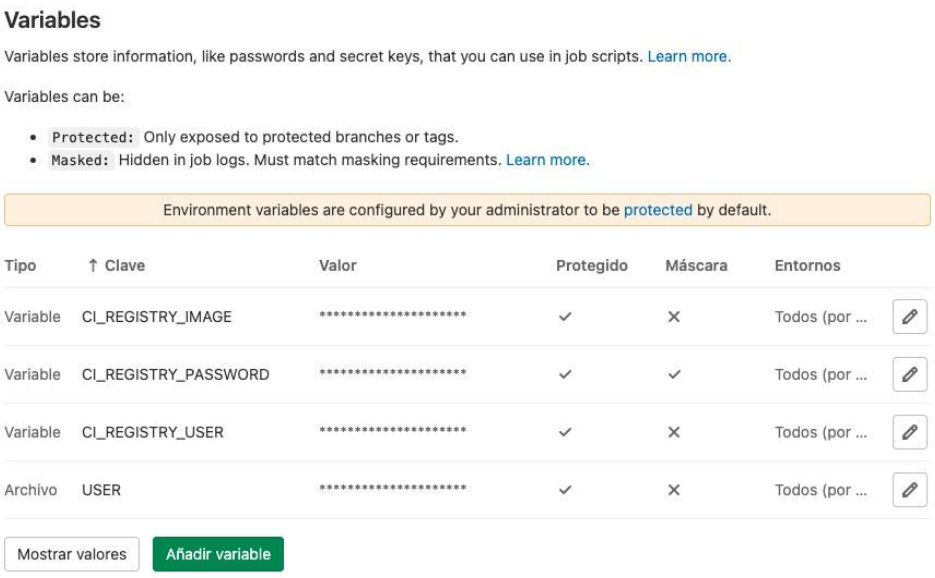


Ilustración 33 Variables de entorno

Las principales variables de entorno que se necesitan crear son las que contienen las credenciales de DockerHub, que son usuario y contraseña, cabe mencionar que todas las variables que son creadas deben estar dentro del proyecto.

Posteriormente se configura la conexión o creación de un Clúster Kubernetes, para la nueva conexión se necesitan los datos que el proveedor del Clúster, principalmente son el Api, el certificado CA y el Token.

Tabla 16 Credenciales Clúster

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURKekNDQWcrZ0F3SUJBZ0lDQm5Vd0RRWUpLb1p
JaHZjTkFRRUxCUUF3TXpFVk1CTUdBMVVFQ2hNTVJHbG4KYVhSaGJFOWpaV0Z1TVJvd0dBWURWU
VFERXhGck9ITmhZ
    server: https://2df1dc32-dc6a-4eb7-896d-c163b12fa43b.k8s.ondigitalocean.com
  name: do-nyc1-cluster
contexts:
- context:
    cluster: do-nyc1-cluster
    user: do-nyc1-cluster-admin
  name: do-nyc1-cluster
```

```
current-context: do-nyc1-cluster
kind: Config
preferences: {}
users:
- name: do-nyc1-cluster-admin
  user:
    token: e1fb141a5878f59bf250b7d19c0d4a3325435345353816044 0d79df9f385cae
```

El certificado CA se decodifica en base64 ya que viene codificado por motivos de seguridad, el mismo se debe ingresar dentro del Gitlab.

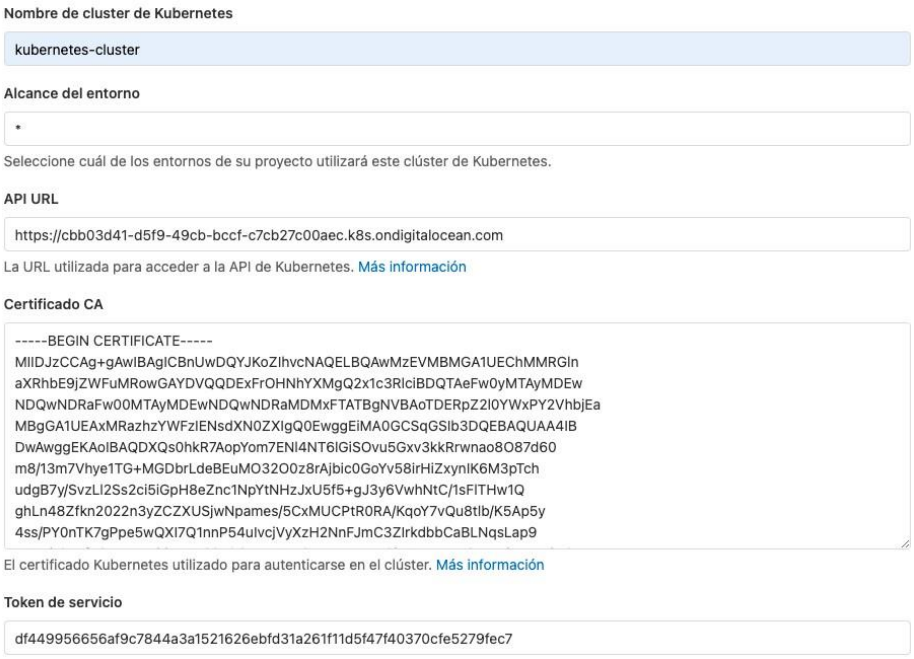


Ilustración 34 Conexión Clúster

Todo lo anterior es simplemente la integración con el Clúster, pero falta un punto importante que ayuda a controlar el Clúster desde Gitlab en cuanto a la construcción y el despliegue de la aplicación, la configuración de los agentes de Gitlab para que realicen toda esta labor, pero en principio no se conoce dónde se ejecutarán, para asegurar que se ejecuten dentro del Clúster se deben eliminar el resto de los Runners o agentes y únicamente dejar el que apunta el Clúster.

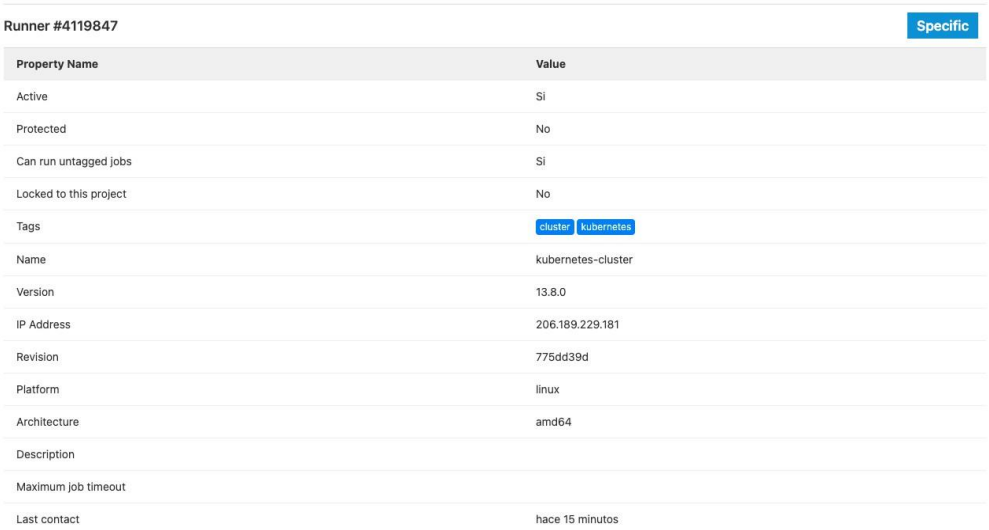


Ilustración 35 Agente Gitlab

De esta forma el proyecto de Gitlab asegura que todos los pasos definidos en el archivo gitlab-ci.yaml se ejecutarán dentro del Clúster.

4.2.3.4 CONFIGURACIÓN CERTIFICADOS SSL

El encargado de recibir y reenviar el tráfico al Ingress será el Loadbalancer, como está de punto de entrada se debe crear 2 registros de tipo A que apunten al dominio y subdominio para Spring y Angular respectivamente.

Ilustración 36 Registros tipo A

Ambos registros apuntan a la misma Ip, podría no tener sentido hacer este proceso, pero recordando que se configura un Ingress que tiene el mismo dominio y subdominio, entonces él será el encargado de apuntar a cada servicio.

Tabla 17 Ingress Hosts

```
- hosts:
    - app-docker.tech
    - api.app-docker.tech
    secretName: certificado-tls
  rules:
  - host: app-docker.tech
    http:
      paths:
      - backend:
          serviceName: frontend-service
          servicePort: 80
  - host: api.app-docker.tech
    http:
      paths:
      - backend:
          serviceName: backend-service
          servicePort: 8080
```

Para la configuración del Ingress simplemente se define el dominio y subdominio que apunten a los servicios que ellos serán encargados de responder a cada petición que llegue por el Loadbancer, cada servicio está explicado en el capítulo anterior.

Actualmente todas las peticiones serán escuchas únicamente por el puerto 80(http), pero el puerto 443(https) no responderá a las peticiones, para agregar los certificados se define un ClusterIssuer el cual estará encargado de emitir los certificados.

```yaml
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: produccion
acme:
  server: https://acme-v02.api.letsencrypt.org/directory
```

Ilustración 37 Objeto ClusterIssuer

Se usa el servidor de producción para evitar el problema que genera la aplicación Frontend, que pide al servidor contar con el protocolo https para realizar las peticiones a los servicios Rest, se especifica que el protocolo ACME apuntando al servidor de producción que Let’s Encrypt utiliza.

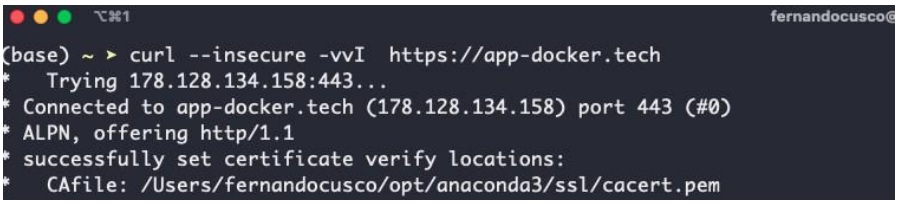


Ilustración 38 Ssl response

4.3 MÉTRICAS DE RENDIMIENTO

Es importante tener en cuenta como están funcionando los servicios en cuestiones de rendimiento, para ello existen varias herramientas para monitorear todo este tipo de métricas, pero las que se han configurado son dos, Prometheus, es quien va a proveer de métricas y va a exponer a través de un servicio, además se hace uso de Grafana el cual está encargada de visualizar todas las métricas recolectadas por Prometheus en múltiples gráficos.

Tabla 18 Deployment Grafana

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
namespace: monitoring
spec:
  replicas: 1
  template:
  metadata:
      name: grafana
      labels:
        app: grafana
    spec:
    containers:
      - name: grafana
        image: grafana/grafana:latest
        ports:
        - name: grafana
          containerPort: 3000
        resources:
          limits:
            memory: "1Gi"
          cpu: "1000m"
          requests:
          memory: 500M
          cpu: "500m"
```

Como primer punto se procede a crear el manifiesto de configuración de tipo Deployment para Grafana y su estado deseado, especificando el número de réplicas, pasado al contendido se tiene la siguiente configuración:

- réplicas: se especifica el número de Pods que van a ejecutar la aplicación en este caso Grafana.
- image: la imagen que ejecutarán los contenedores.
- containerPort: es el puerto que exponen los contenedores y, por ende, es el puerto de Grafana.
- limits: se especifica los recursos asignados a cada contendor en capacidad máxima.
- requests: se especifica los recursos mínimos que debe tener cada contendor.

Toda la configuración ha sido definida en el manifiesto, aún los contendores no pueden ser accedidos desde fuera, para resolver esto se tiene dos formas, la primera asignando el Deployment a un servicio y la segunda es exponer el puerto del Pod a través de un puerto de la máquina local, esta forma sirve más que nada para realizar depuraciones, no se recomienda usarla en entornos de producción, puesto que el Pod como se explicó anteriormente es efímero y el Clúster puede llegar a eliminarlo en cualquier momento.

Tabla 19 Servicio Grafana

```
apiVersion: v1
kind: Service
metadata:
  name: grafana
namespace: monitoring
spec:
  selector:
    app: grafana
  type: NodePort
  ports:
    - port: 3000
      targetPort: 3000
      nodePort: 32000
```

Para el servicio es un poco más simple, ya que solo se definen los puertos de reenvío y el tipo de servicio que es NodePort para este caso, el puerto será aleatorio, aunque se puede definir uno especifico, el selector que hará match con los Pods tiene la etiqueta “app: grafana” y expone a través de un único Endpoint. Para Prometheus es prácticamente igual, la diferencia es que la imagen de Docker es “prom/prometheus” y el puerto que por defecto es 9090.

Finalmente, la forma de exponer será de la forma no convencional, usando el comando “port-forward”, como se explicó, expone el Pod, puede ser cualquiera en el caso que se tengan más de una réplica, lo importante es especificar al final el puerto, recordemos que toma un puerto de la máquina local y lo mapea al puerto de Pod para que se puede acceder a través de localhost.

El comando complete es el siguiente “kubectl port-forward pod/prometheus-7c78857f5c-7dq7h 9090”.

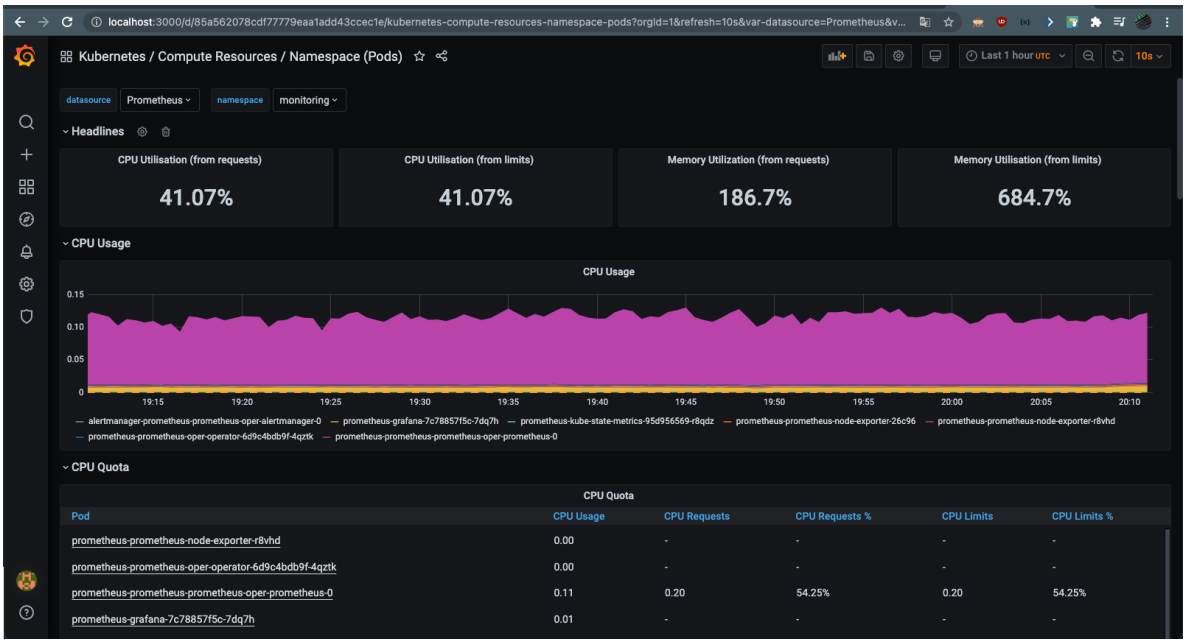


Ilustración 39 Grafana métricas CPU

4.4 IMPLEMENTACIÓN DE PRUEBA DE CARGA

Se ha desarrollado una prueba de carga con respecto al servidor que ejecuta la aplicación con Spring, la idea principal es conocer el número de peticiones que puede responder de manera simultánea cuando el cliente envíe una solicitud al servidor.

Para ello se definió una prueba a un Endpoint específico, https://api.docker.tech/token/generate-token, este Endpoint es de tipo Post, por donde se debe pasar un objeto de tipo Json como cuerpo de la petición. Por ejemplo:

```
{
   "email": "usuario@mail.com",
   "enable": true,
   "password": "my-key@345EXC"
}
```

Usando la herramienta JMeter, se ha definido una prueba que contiene el número de peticiones, el Endpoint al cual se enviará la petición y el cuerpo de la petición, lo que el servidor debe responder, para este caso será un Token de acceso.

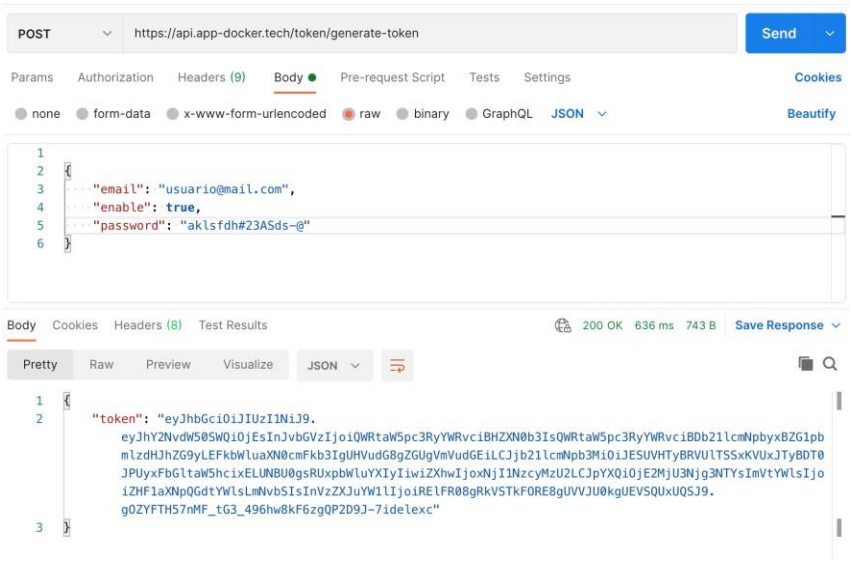


Ilustración 40 Respuesta servidor

Las pruebas se pueden ejecutar a mano, pero lo ideal sería automatizar las pruebas y tener un control central dentro del Pipeline que se definió anteriormente, la idea es que cuando se suba un nuevo cambio ejecutar las pruebas como punto final, sin importar si existe un error, para evitar

que los cambios no se apliquen, entonces, para lograr todo esto se tiene que definir un nuevo contendor dentro del Clúster que ejecute las pruebas y luego de ello eliminar el contenedor. Cambiando el Pipeline para ejecutar las pruebas se implementó tres pasos, el primero se encarga de crear el contendor y ejecutar las pruebas, la segunda muestra la salida de las pruebas, como el tiempo de respuesta, si existe errores por cada petición, entre varios más y finalmente el tercero se encarga de eliminar toda la configuración con relación al contendor en el que se realizaron las pruebas.

Tabla 20 Pipeline Test

```
stress:
  stage: stress
  image:
    name: bitnami/kubectl
    entrypoint: [""]
  script:
  - kubectl -n default apply -f https://raw.githubusercontent.com/Fernando-Cusco/stress/main/deployment.yaml
  only:
  - master
logs:
  stage: logs
  image:
    name: bitnami/kubectl
    entrypoint: [""]
  script:
  - kubectl -n default logs -l app=test
  only:
  - master
clean:
  stage: clean
  image:
    name: bitnami/kubectl
    entrypoint: [""]
  script:
  -  kubectl -n default delete -f https://raw.githubusercontent.com/Fernando-Cusco/stress/main/deployment.yaml
  only:
  - master
```

Con todos estos nuevos cambios para las pruebas el Pipeline cambia, pasando de tener dos pasos a tener cinco.

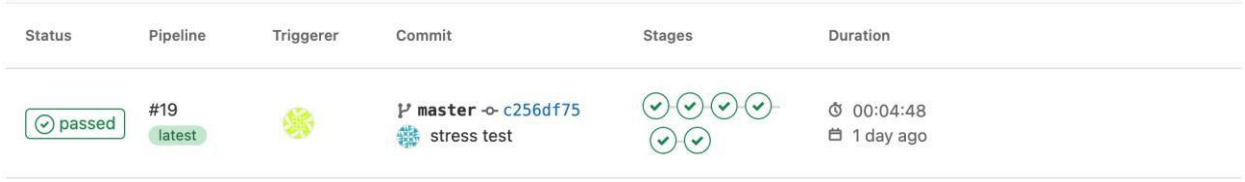


Ilustración 41 Actualización Pipeline

4.5 IMPLEMENTACIÓN PRUEBA DE STRESS

La prueba de Stress está enfocada al Clúster, se pone a prueba el número de réplicas máximas que puede soportar el Clúster para un grupo de contendores o para un único contenedor el cual va a estar realizando peticiones constantes al Endpoint de la aplicación Web, la idea principal es verificar que tantas cargas soportaría, teniendo en cuenta que solo cuenta con 2 nodos con características de Hardware básicos.

Tabla 21 Stress Test

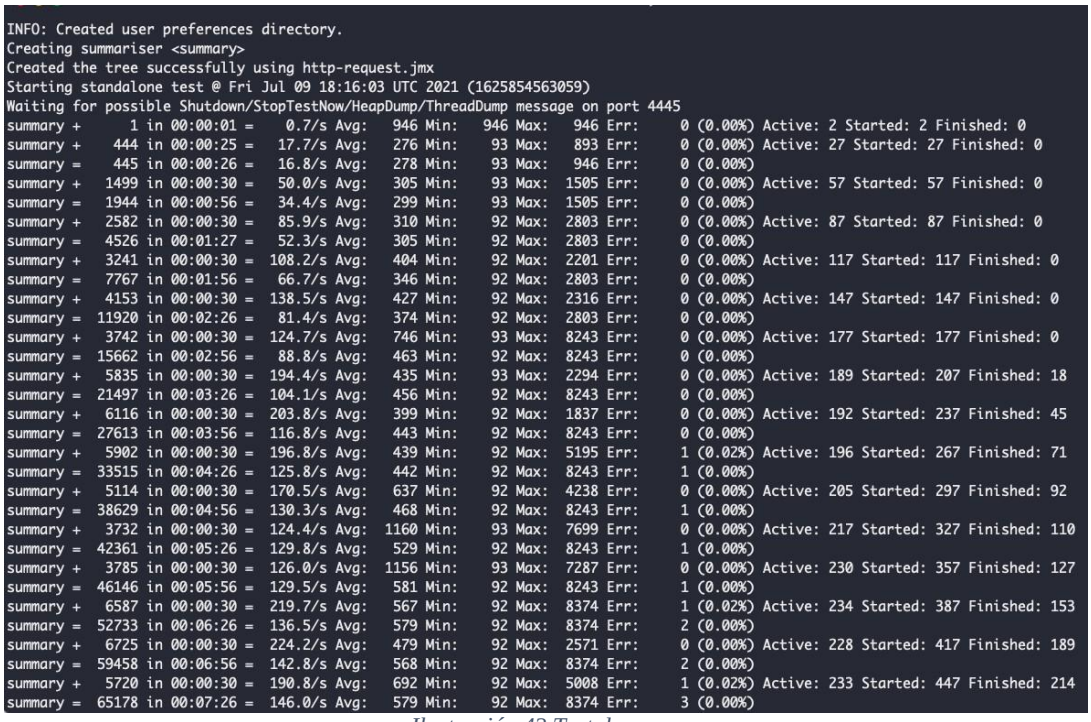
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: stress
  labels:
    app: stress
namespace: default
spec:
  template:
    spec: containers:
      image: busybox
      command:
        - /bin/sh
        - -c
        - "while true; do wget -q -O- https://app-docker.tech; done"
```

La prueba trata de un contendor simple, que ejecuta una petición a "https://app-docker.tech", cuenta con una sola replica que posteriormente se escalara a un determinado número de replicas, para que de esta manera se ponga a prueba el límite que soporta el Clúster.

4.6 RESULTADOS

A lo largo de la implementación del proyecto se logró obtener varios resultados que demuestran el cumplimiento de todos los objetivos específicos, cada objetivo se lo ha ido desarrollando durante todo el proceso, en este punto se trata de analizar cuáles fueron los resultados obtenidos con respecto a los requerimientos no funcionales. Entonces, teniendo en cuenta los requerimientos más importantes, a continuación, se muestra los resultados obtenidos.

Ejecución de las pruebas de carga con respecto al servidor utilizando Jmeter.



```
INFO: Created user preferences directory.
Creating summariser <summary>
Created the tree successfully using http-request.jmx
Starting standalone test @ Fri Jul 09 18:16:03 UTC 2021 (1625854563059)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump message on port 4445
summary +      1 in 00:00:01 =    0.7/s Avg:   946 Min:   946 Max:   946 Err:     0 (0.00%) Active: 2 Started: 2 Finished: 0
summary +    444 in 00:00:25 =   17.7/s Avg:   276 Min:    93 Max:   893 Err:     0 (0.00%) Active: 27 Started: 27 Finished: 0
summary =    445 in 00:00:26 =   16.8/s Avg:   278 Min:    93 Max:   946 Err:     0 (0.00%)
summary +   1499 in 00:00:30 =   50.0/s Avg:   305 Min:    93 Max:  1505 Err:     0 (0.00%) Active: 57 Started: 57 Finished: 0
summary =   1944 in 00:00:56 =   34.4/s Avg:   299 Min:    93 Max:  1505 Err:     0 (0.00%)
summary +   2582 in 00:00:30 =   85.9/s Avg:   310 Min:    92 Max:  2803 Err:     0 (0.00%) Active: 87 Started: 87 Finished: 0
summary =   4526 in 00:01:27 =   52.3/s Avg:   305 Min:    92 Max:  2803 Err:     0 (0.00%)
summary +   3241 in 00:00:30 =  108.2/s Avg:   404 Min:    92 Max:  2201 Err:     0 (0.00%) Active: 117 Started: 117 Finished: 0
summary =   7767 in 00:01:56 =   66.7/s Avg:   346 Min:    92 Max:  2803 Err:     0 (0.00%)
summary +   4153 in 00:00:30 =  138.5/s Avg:   427 Min:    92 Max:  2316 Err:     0 (0.00%) Active: 147 Started: 147 Finished: 0
summary =  11920 in 00:02:26 =   81.4/s Avg:   374 Min:    92 Max:  2803 Err:     0 (0.00%)
summary +   3742 in 00:00:30 =  124.7/s Avg:   746 Min:    93 Max:  8243 Err:     0 (0.00%) Active: 177 Started: 177 Finished: 0
summary =  15662 in 00:02:56 =   88.8/s Avg:   463 Min:    92 Max:  8243 Err:     0 (0.00%)
summary +   5835 in 00:00:30 =  194.4/s Avg:   435 Min:    93 Max:  2294 Err:     0 (0.00%) Active: 189 Started: 207 Finished: 18
summary =  21497 in 00:03:26 =  104.1/s Avg:   456 Min:    92 Max:  8243 Err:     0 (0.00%)
summary +   6116 in 00:00:30 =  203.8/s Avg:   399 Min:    92 Max:  1837 Err:     0 (0.00%) Active: 192 Started: 237 Finished: 45
summary =  27613 in 00:03:56 =  116.8/s Avg:   443 Min:    92 Max:  8243 Err:     0 (0.00%)
summary +   5902 in 00:00:30 =  196.8/s Avg:   439 Min:    92 Max:  5195 Err:     1 (0.02%) Active: 196 Started: 267 Finished: 71
summary =  33515 in 00:04:26 =  125.8/s Avg:   442 Min:    92 Max:  8243 Err:     1 (0.00%)
summary +   5114 in 00:00:30 =  170.5/s Avg:   637 Min:    92 Max:  4238 Err:     0 (0.00%) Active: 205 Started: 297 Finished: 92
summary =  38629 in 00:04:56 =  130.3/s Avg:   468 Min:    92 Max:  8243 Err:     1 (0.00%)
summary +   3732 in 00:00:30 =  124.4/s Avg:  1160 Min:    93 Max:  7699 Err:     0 (0.00%) Active: 217 Started: 327 Finished: 110
summary =  42361 in 00:05:26 =  129.8/s Avg:   529 Min:    92 Max:  8243 Err:     1 (0.00%)
summary +   3785 in 00:00:30 =  126.0/s Avg:  1156 Min:    93 Max:  7287 Err:     0 (0.00%) Active: 230 Started: 357 Finished: 127
summary =  46146 in 00:05:56 =  129.5/s Avg:   581 Min:    92 Max:  8243 Err:     1 (0.00%)
summary +   6587 in 00:00:30 =  219.7/s Avg:   567 Min:    92 Max:  8374 Err:     1 (0.02%) Active: 234 Started: 387 Finished: 153
summary =  52733 in 00:06:26 =  136.5/s Avg:   579 Min:    92 Max:  8374 Err:     2 (0.00%)
summary +   6725 in 00:00:30 =  224.2/s Avg:   479 Min:    92 Max:  2571 Err:     0 (0.00%) Active: 228 Started: 417 Finished: 189
summary =  59458 in 00:06:56 =  142.8/s Avg:   568 Min:    92 Max:  8374 Err:     2 (0.00%)
summary +   5720 in 00:00:30 =  190.8/s Avg:   692 Min:    92 Max:  5008 Err:     1 (0.02%) Active: 233 Started: 447 Finished: 214
summary =  65178 in 00:07:26 =  146.0/s Avg:   579 Min:    92 Max:  8374 Err:     3 (0.00%)
```

Ilustración 42 Test de carga

Examinando cada muestra del JMeter en las pruebas, se tiene que, en términos generales por cada grupo de peticiones el porcentaje de error no es mayor a 0.05%, teniendo en cuenta que con cada iteración el número de peticiones aumenta. Además, los tiempos mínimos que están medidos en milisegundos no superan un segundo por cada grupo de muestras y como tiempos máximos en la ilustración 42 se tiene una muestra con 8374 milisegundos para 65178 peticiones, los procesos

activos, iniciados y finalizados dependiendo el número de hilos especificados deberán completarse según vaya transcurriendo el tiempo que le tome a JMeter realizar la prueba.

Analizando las pruebas, pero ahora de lado de Grafana se tiene varios resultados. Dependiendo del uso de memoria, CPU, ancho de banda, entro varias más. Escalando el número de contendores de la prueba de Stress a 60 se verán reflejados altos consumos en cuanto a memoria y CPU. Antes de incrementar las réplicas el estado general es como se muestra en la siguiente ilustración.

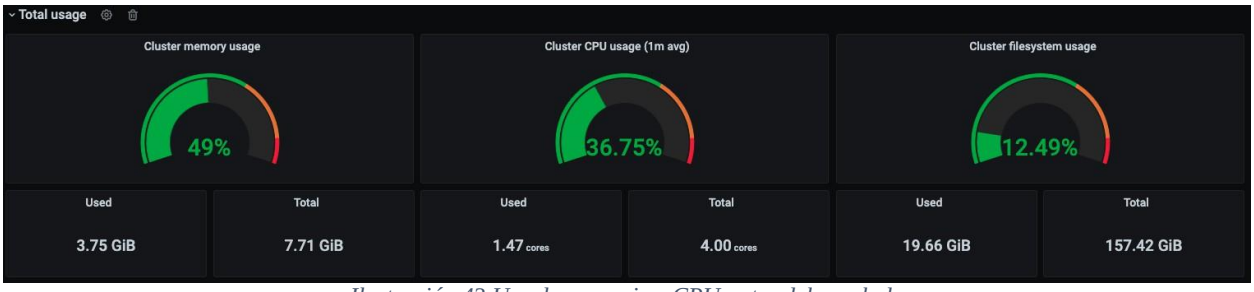


Ilustración 43 Uso de memoria y CPU antes del escalado

Luego de aproximadamente un minuto los contenedores escalaron al número de réplicas establecidas, mostrando cambios en cuestiones de consumo, principalmente en cuestiones de CPU ya que crece el número de contendores, para el caso de la memoria no hubo un consumo mayor, puesto que se trata de contendores que ejecutan un Linux básico.

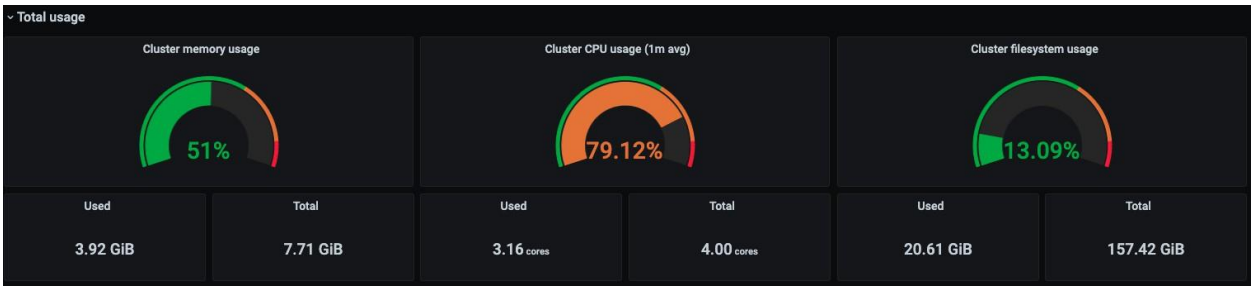


Ilustración 44 Uso de memoria y CPU luego del escalado

Ahora bien, analizando los resultados que Jmeter dio a través de su interfaz gráfica son varios, los cuales son muy similares a los que el contenedor arrojo.

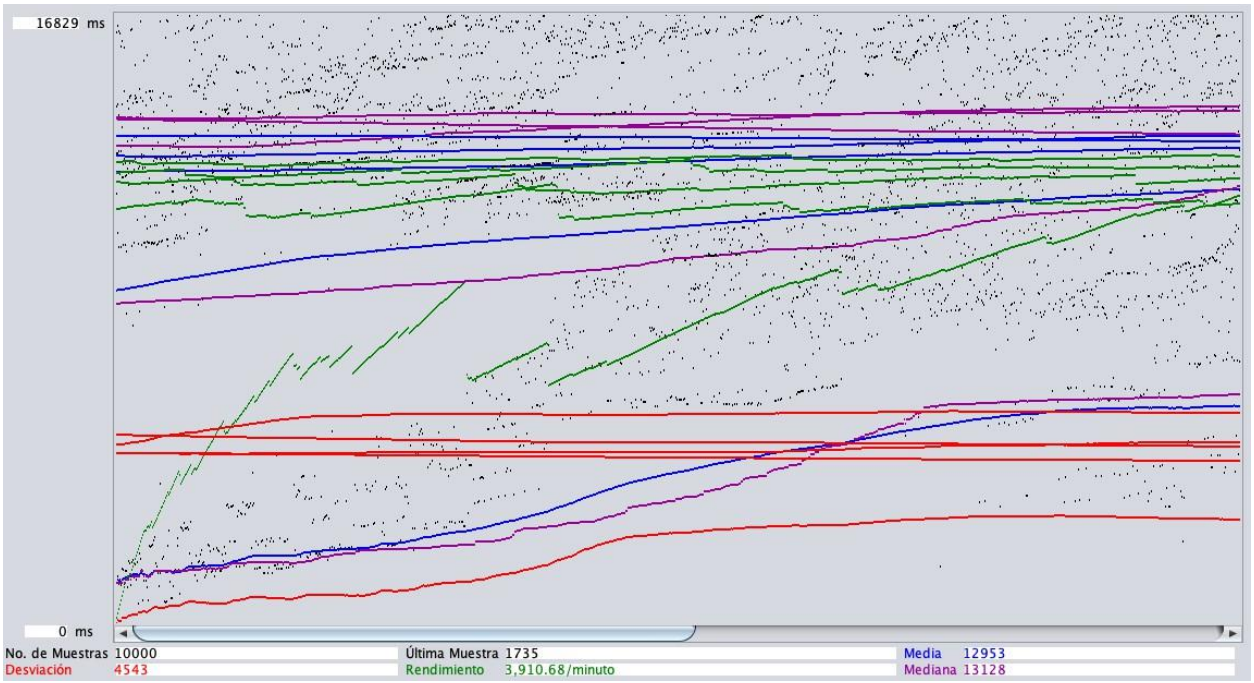


Ilustración 45 Grafica Jmeter

Hasta el término de la ejecución el número total de solicitudes realizadas a la aplicación Web son 10000, el tiempo de respuesta de la última petición es de 1735ms, lo cual muestra un buen tiempo de respuesta, recordando que las peticiones son de manera simultánea. Como es común en todo sistema el tiempo de respuesta dependiendo el número de conexiones va variando para el caso de la desviación se tiene 4543 ms, la cantidad de peticiones con éxito hacia el servidor son aproximadamente de 3900 en 1 minuto, para un grupo de peticiones la media es aproximadamente de 12 segundos, el 50% de todas las peticiones no superaron los 13 segundos, teniendo en cuenta que el número total de procesos son de 1000 conexiones el tiempo de respuesta general como máximo llega a superar los 4 segundos.

Analizando el tiempo de respuesta por las peticiones o el grupo de peticiones que se realiza durante el intervalo de tiempo, se tiene un resultado muy similar, el tiempo de respuesta general es muy similar al mostrado en los resultados analizados de la ilustración 45 y 42, lo que cambia que es una ejecución nueva.

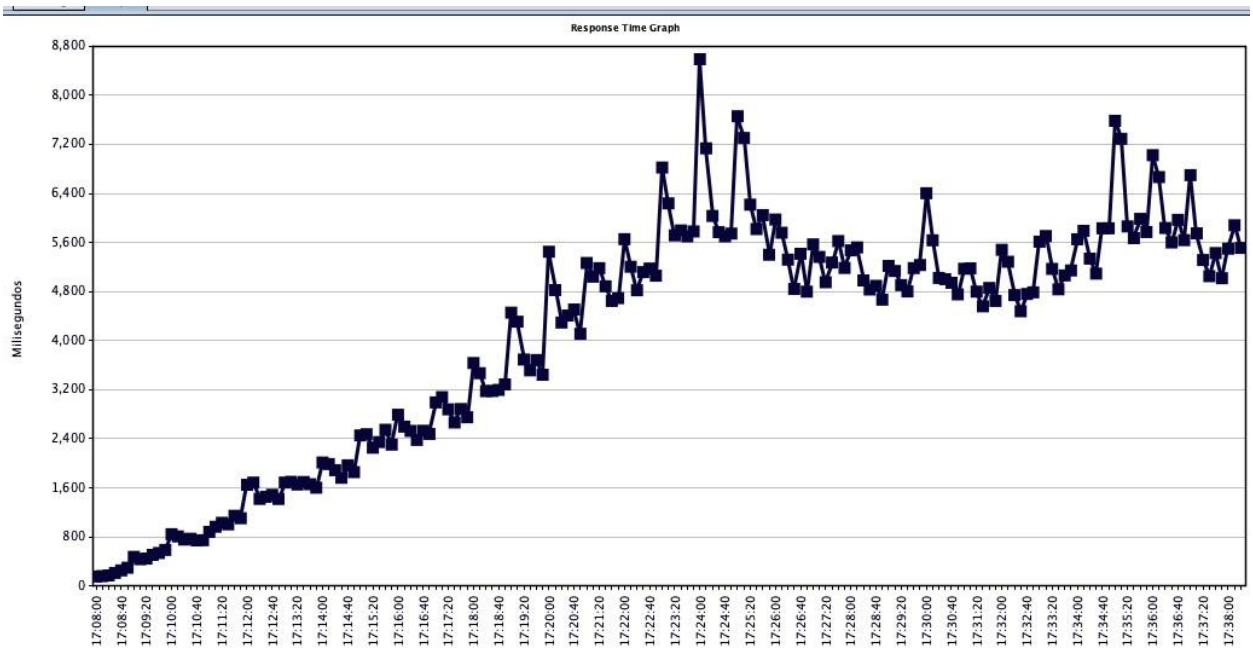


Ilustración 46 Tiempo de respuesta

4.7 IMPLEMENTACIÓN PARA UN PROYECTO NUEVO

En este apartado se explica de manera general la forma de como generar una arquitectura similar que contenga una aplicación, este proyecto es uno que está desarrollado con JavaEE que será desplegado en un servidor de aplicaciones Wildfly. Antes de empezar se debe generar el archivo War que contiene la aplicación, para este punto hay dos opciones, la una usar Docker con una imagen de Maven y la otra general de forma manual el archivo War, para este ejemplo se utiliza un contenedor Docker para generar el archivo War, entonces para ello se debe ejecutar comando que se muestra en la ilustración 47. Lo que el comando realiza en primera instancia es buscar de manera local la imagen de Maven, caso contrario la bajara del repositorio oficial y a continuación ejecutara el comando “mvn clean install” para compilar el proyecto y generar el archivo War.

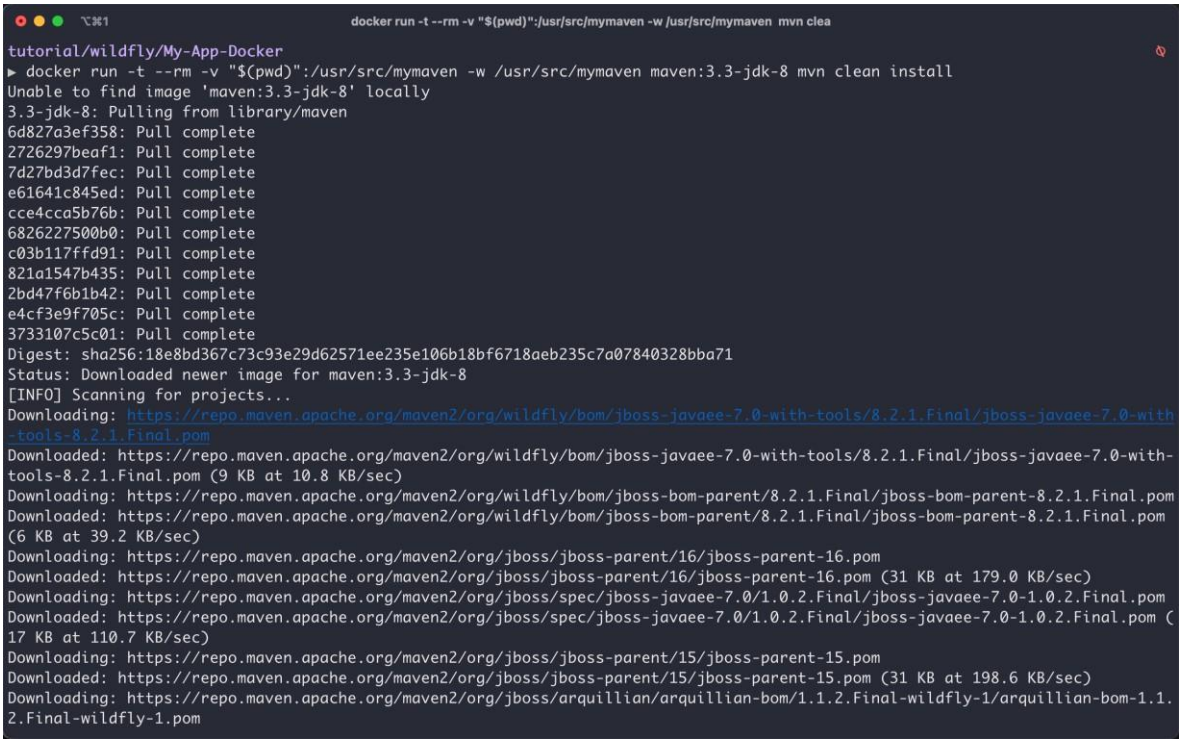


Ilustración 47 Maven install usando Docker

Si el proyecto base no contiene errores el proceso debe dar como resultado el directorio “target” el mismo que contiene el archivo War. Ahora bien, el siguiente paso es empaquetar todo dentro de una imagen, la imagen debe contener lo siguiente, el archivo War, el archivo de configuración (standalone.xml) y los módulos para la conexión hacia la base de datos, de esta manera el Dockerfile debe ser similar a la siguiente ilustración.

```
FROM jboss/wildfly:20.0.1.Final
COPY org /opt/jboss/wildfly/modules/org
COPY standalone.xml /opt/jboss/wildfly/standalone/configuration
ADD My-App-Docker/target/My-App-Docker.war /opt/jboss/wildfly/standalone/deployments/
RUN /opt/jboss/wildfly/bin/add-user.sh admin Admin#123 --silent
EXPOSE 9990
EXPOSE 8080
USER jboss
CMD [ "/opt/jboss/wildfly/bin/standalone.sh", "-b", "0.0.0.0", "-bmanagement", "0.0.0.0" ]
```

Ilustración 48 Dockerfile Wildfly

El Dockerfile contiene los pasos necesarios para construir la imagen personalizada, empezando con la imagen base de Wildfly en la versión 20.0.1.Final, a continuación se copian los módulos

que son para la base de datos en este caso Postgresql, el archivo de configuración para Wildfly y se agrega el archivo War al directorio “deployments”, luego se crea un usuario para la administración del servidor, además se exponen los puertos para el acceso a la aplicación y otro para la administración y finalmente, se define el comando que se ejecutara la primera vez que inicie el servidor que es para arrancar el servidor que contiene la aplicación agregada.

Para construir la imagen se debe ejecutar el comando que se muestra en la ilustración 49, el siguiente parámetro “-t fcusco96/my-wildfly:v1” indica que la imagen va a ser etiquetada con el nombre que se especifica.

Ilustración 49 Docker build

El resultado debe ser una imagen con el nombre del parámetro indicado, que estará alojada en el Docker de la maquina local.

Ahora toca hacer uso Kubernetes para crear la arquitectura que va a contener la base de datos, la aplicación con JavaEE todo esto a través de manifiestos declarativos.

Analizando un poco de la estructura para la base de datos, como se muestra en la ilustración 50, se tiene que, la imagen base es una de Postgresql con la versión 10, contara únicamente con una réplica, de igual manera se definen las variables de entorno las mismas que contienen las credenciales de la base de datos para la conexión, además se agrega el apartado de recursos, el cual

permite el Pod arrancar con un mínimo de recurso de memoria y de Cpu, hasta un máximo de uso de memoria y Cpu, el Pod no podrá superar los límites establecidos y finalmente se expone el puerto que por defecto es 5432 para Postgresql.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-deployment
  labels:
    app: ups
    db: postgres
spec:
  selector:
    matchLabels:
      app: ups
      db: postgres
  replicas: 1
  template:
    metadata:
      labels:
        app: ups
        db: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:10
          env:
            - name: POSTGRES_USER
              value: admin
            - name: POSTGRES_PASSWORD
              value: admin
            - name: POSTGRES_DATABASE
              value: app
          ports:
            - containerPort: 5432
          resources:
            requests:
              memory: "64Mi"
              cpu: "250m"
            limits:
              memory: "128Mi"
              cpu: "500m"
```

Ilustración 50 Configuración de la base de datos.

Para la parte del servidor es muy similar al manifiesto de la base de datos, con dos diferencias la una es el uso de la imagen personalizada que en el primer punto se la construyo de manera local y la otra es el puerto expuesto, en la siguiente ilustración se muestra el manifiesto completo.

```yaml
kind: Deployment
metadata:
  name: java-app
  labels:
    app: java
spec:
  selector:
    matchLabels:
      app: java
  replicas: 5
  template:
    metadata:
      labels:
        app: java
    spec:
      containers:
        - name: java-app
          image: fcusco96/my-wildfly:v1
          ports:
            - containerPort: 8080
```

Ilustración 51 Manifiesto servidor

Para levantar los servicios definidos basta con ejecutar el siguiente comando “kubectl apply -f .”, toma todos los archivos yaml que están dentro del directorio y crea los recursos como lo son la base de datos y la aplicación en este caso. Si desea conocer más acerca del proyecto, puede acceder al proyecto completo usando el siguiente enlace https://github.com/Fernando-Cusco/javaee-kubernetes.

CAPÍTULO 5 CONCLUSIONES, RECOMENDACIONES Y TRABAJOS FUTUROS

5.1 CONCLUSIONES

Se ha conseguido lograr el objetivo principal que es automatizar las tareas que son necesarias para realizar el proceso de construcción y despliegue de una aplicación (DevOps), durante el desarrollo se obtuvo una gran ventaja con la integración de cada una de las herramientas seleccionadas, empezando por Docker que tiene un papel importante para generar una imagen que contenga la aplicación a desplegarse conjuntamente con su configuración como imagen base, puertos, y scripts extras que permiten el correcto funcionamiento de una aplicación, sin importar el entorno de contenedores o entorno de Kubernetes en el que va a ejecutarse, esto genera cierta confianza.

Kubernetes tuvo el papel más importante, ya que permitió gestionar los contenedores, pero a gran escala, de esta manera la aplicación siempre está activa, permitiendo generar réplicas para que respondan a las peticiones de los clientes de manera balanceada, gracias a su simpleza de configuración a través de manifiestos los cuales permiten controlar de mejor manera el despliegue de la aplicación a través de las diferentes formas que tiene para exponer los Deployments, los 2 más importantes que se utilizaron fueron ClusterIp y LoadBalancer aportando utilidad para dar acceso a los usuarios.

Gitlab CI permite gestionar la integración continua y el despliegue continuo, ya que se configuró cada instrucción necesaria para conectarse al Clúster de Kubernetes y realizar todo el proceso desde la construcción de la imagen, los Test y la actualización del Pod(aplicación), otro tema importante que se tomó en cuenta es que cada aplicación tenga su propio certificado seguro(Ssl), la herramienta que encajo perfectamente con el Clúster es LetsEncrypt, que toda la configuración está basada en manifiestos propios de Kubernetes y que además de esto son gratuitos, autofirmado de claves, con renovación automática, para el caso de no hacer uso de renovación automática el

servidor emite avisos con un periodo de entre 20 y 30 días que los certificados están por vencer y lo mejor, que permite hacer uso de un servidor de pruebas propio que disponen para evitar posibles bloqueos al dominio, luego de la configuración y funcionamiento se cambió al servidor de producción, Por otro lado, es necesario tener un dominio y un proveedor de entorno Cloud, en base a ello se utilizó DigitalOcean ya que permitió el manejo completo de la arquitectura y la integración del balanceador de carga que apuntan a los Ingress del Clúster de Kubernetes y el control total de dominio Dns, además que los precios para desarrollar son económicos a comparación de otros proveedores. Por otro lado es muy importante definir pruebas, como resultados de las pruebas realizadas al servidor y al cliente, se obtuvieron resultados muy favorables en cuanto a tiempos de respuesta que no sobrepasan los 4.5ms, porcentajes de errores bajos por cada grupo de peticiones, con un máximo de 0.08% de errores que representan 8 peticiones de las 10000. Recordando que las pruebas llaman al servicio correspondiente, en este caso el contenedor que contiene el servidor responderá a todas las peticiones generando nuevos niveles de consumos en cuanto a memoria, ancho de banda, paquetes recibidos, enviados. La prueba de Stress de igual manera mantiene hasta un máximo de 75% de uso de recursos de todo el Clúster en un rango de escalado de contendores de 30, 40 y 50 para la prueba de Stress, respondiendo de manera correcta las peticiones que se realizan dentro de cada uno. Todo esto se ven perfectamente reflejado en Grafana utilizando las propias métricas que provee y de esta manera corroborar que las pruebas se ejecuten de acuerdo con el servicio llamado. En cuanto a los requerimientos funcionales y no funcionales se lograron completar sin mayor dificultad, de igual manera los resultados son muy favorables en términos generales para lo que es la arquitectura, el despliegue y la integración continuos, hasta el momento no se han generado mayores problemas en cuestiones de rendimiento. Sin embargo, se debe tener

en cuenta que el Clúster en el que esta desplegado toda la arquitectura cuenta con características de Hardware básicas para desarrollo y no para producción.

Finalmente, la forma de cómo se fue desarrollando cada objetivo, se enfoca principalmente en dos partes. La primera que es el uso e implementación de tecnologías aprendidas durante los últimos ciclos universitarios, por ejemplo, las herramientas para realizar los diferentes tipos de prueba, se recurrió a analizar más que nada los proyectos donde anteriormente ya se había intentado solucionar problemas similares a diferencia que ahora se lo aplica dentro de un entorno basado en contenedores que básicamente es lo que cambió. La segunda trata de la recopilación información necesaria para realizar el objetivo principal que es la automatización, dicha información se consiguió de múltiples fuentes, pero más que nada todo el aprendizaje del estado del arte se lo adquirió en cursos, talleres sobre manejo de contendores como los son Docker, Kubernetes y lo que es automatización, luego de ello con toda la información adquirida se fue desarrollando cada objetivo ajustando a lo que se necesita cumplir.

5.2 RECOMENDACIONES

Es muy importante tener claro la arquitectura que se maneje, pero al momento de desplegar toda la arquitectura se recomienda usar herramientas que funcionen de manera local, por ejemplo, la versión liviana de Kubernetes como K3s, MicroK8s o incluso Minikube, la ventaja que se tiene es que cada uno provee un Clúster listo para trabajar y lo mejor que acepta los mismos manifiestos que funcionan en la versión de Kubernetes para producción, con esto se puede desarrollar todos los tipos de pruebas a nivel de Clúster y depurar errores que se tengan a nivel de sintaxis, errores con respecto a los contenedores, construcciones de imágenes o incluso despliegues de aplicaciones. Otra ventaja importante es que estas herramientas a pesar de ser usadas para desarrollo cuenta con su propio sistema que maneja el uso de múltiples nodos, dependiendo de las

necesidades el Clúster ajustara las configuraciones especificadas al número de nodos que se hayan definido, también a tener en cuenta que si se despliega una aplicación en el entorno de desarrollo es seguro que la misma aplicación se desplegara en el entorno de producción con una versión para servidor de Kubernetes, por tal motivo se recomienda usar la versión para desarrollo que dispone Kubernetes para probar y hacer Debug de las aplicaciones, por otro lado se debe tener presente que para ejecutar Pipelines que sean principalmente de construcción de imágenes, test o despliegues, se debe tener un mínimo de 2 a 3 nodos dentro del Clúster, puesto que las tereas no pueden ejecutarse totalmente. Si bien durante el desarrollo del proyecto para contrastar el funcionamiento correcto de la aplicación desplegada se ejecutan 2 pruebas, es importante tener en cuenta que se tiene varias más, por tal motivo se recomienda ejecutar pruebas extra como lo son de resistencia y de escalamiento bien sea a nivel de aplicación o nivel de instancias, tener en cuenta que las pruebas dicen mucho de sí, ya que son una visión a futuro de lo que puede pasar cuando exista un tráfico mayor a uno que sea menor.

5.3 TRABAJOS FUTUROS

En el presente trabajo se propuso una arquitectura con ciertas tecnologías con las que lanzamos una implementación de DevOps, pero existen otras líneas en las que se puede mejorar la arquitectura con nuevas tecnologías, optimizar los tiempos de entrega y generar un valor agregado a los clientes.

A continuación, se menciona posibles trabajos futuros:

- Diseño de una arquitectura para integración y despliegue continuos de aplicaciones Web desde Github Actions con Docker y Kubernetes administrado con Rancher.

- Diseñar y desplegar una arquitectura en la nube (Google, DigitalOcean, Aws, Azure) basada en código usando Terraform para automatizar el desarrollo continuo de aplicaciones con Docker y Jenkins.

- Mejora continua para sistemas distribuidos y arquitecturas en la nube, automatizando fallos causados aleatoriamente con la herramienta Chaos Monkey.

REFERENCIAS

Walls, M. (2015). Building a DevOps Culture-DevOps is as much about culture as it is about tools, 1st edn. 2nd release.

Saito, H., Lee, H. C. C., & Wu, C. Y. (2019). DevOps with Kubernetes: accelerating software delivery with container orchestrators. Packt Publishing Ltd.

Dompablo Tobar, J. (2018). DevOps para automatización de Gitlab en alta disponibilidad (Bachelor's thesis).

Burns, B., Beda, J., & Hightower, K. (2018). Kubernetes. Dpunkt.

Smith, R. (2017). Docker Orchestration. Packt Publishing Ltd.

Arefeen, M. S., & Schiller, M. (2019). Continuous Integration Using Gitlab. Undergraduate Research in Natural and Clinical Science and Technology Journal, 3, 1-6.

Dinesh, S. (2018). Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?. Marzo de 2018.

Medel, V., Rana, O., Bañares, J. Á., & Arronategui, U. (2016, December). Modelling performance & resource management in kubernetes. In Proceedings of the 9th International Conference on Utility and Cloud Computing (pp. 257-262).

Medel, V., Tolosana-Calasanz, R., Bañares, J. Á., Arronategui, U., & Rana, O. F. (2018). Characterising resource management performance in Kubernetes. Computers & Electrical Engineering, 68, 286-297.

Nagarajan, A. D., & Overbeek, S. J. (2018, October). A DevOps implementation framework for large agile-based financial organizations. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"* (pp. 172-188). Springer, Cham.

Sommerville, I. (2005). Requerimientos del software. *Ingeniería del software, 7a ed., PEARSON EDUCACIÓN, Madrid, SPA*, 109-110.

Babar, Z., Lapouchnian, A., & Yu, E. (2015, November). Modeling DevOps deployment choices using process architecture design dimensions. In *IFIP Working Conference on The Practice of Enterprise Modeling* (pp. 322-337). Springer, Cham.

Chaves, M. A. (2005). La ingeniería de requerimientos y su importancia en el desarrollo de proyectos de software. *InterSedes: Revista de las Sedes Regionales*, *6*(10), 1-13.

Cois, A. (30 de abril de 2015). DevOps Case Study: Netflix and the Chaos Monkey. Obtenido de: https://insights.sei.cmu.edu/blog/devops-case-study-netflix-and-the-chaos-monkey/

Pal, T. (15 de mayo de 2018). Focusing on the DevOps Pipeline, Obtenido de: https://www.capitalone.com/tech/software-engineering/focusing-on-the-devops-pipeline/

Gupta, V., Kapur, P. K., & Kumar, D. (2017). Modeling and measuring attributes influencing DevOps implementation in an enterprise using structural equation modeling. *Information and software technology*, *92*, 75-91.

Chawla, H., & Kathuria, H. (2019). Monitoring Azure Kubernetes Service. In *Building Microservices Applications on Microsoft Azure* (pp. 179-191). Apress, Berkeley, CA.

Ionos. (14 de octubre de 2019). Infrastructure as Code. Obtenido de: https://www.ionos.es/digitalguide/servidores/know-how/infrastructure-as-code/

Docker. (2020). Orientation and setup. Obtenido de: https://docs.docker.com/get-started/

Docker. (2020). Docker architecture. Obtenido de: https://docs.docker.com/get-started/overview/

Robin, M. (19 de octubre de 2018). 9 Reasons DevOps Is Better With Docker and Kubernetes. Obtenido de: https://dzone.com/articles/9-reasons-why-devops-is-better-with-docker-amp-kub

Redhat. (2020). What is Docker. Obtenido de: https://www.redhat.com/es/topics/containers/what-is-docker

K3s. (2020). Lightweight Kubernetes. Obtenido de: https://k3s.io/

Let’s Encrypt. (2020). Getting Started. Obtenido de: https://letsencrypt.org/getting-started/

Kubernetes. (2020). What is Ingress?. Obtenido de: https://kubernetes.io/docs/concepts/services-networking/ingress/

Grafana. (2020). Architecture. Obtenido de: https://grafana.com/docs/tempo/latest/operations/architecture/

Prometheus. (2020). Prometheus for developers. Obtenido de: https://awesomeopensource.com/project/danielfm/prometheus-for-developers

Jenkins. (2020). Jenkins User Documentation. Obtenido de: https://www.jenkins.io/doc/

Travis. (2020). Core Concepts for Beginners. Obtenido de: https://docs.travis-ci.com/user/for-beginners/

Gitlab. (2020). Gitlab CI/CD. Obtenido de: https://docs.gitlab.com/ee/ci/#gitlab-cicd-concepts

Gitlab. (2020). Gitlab Runner. Obtenido de https://docs.gitlab.com/runner/

Gitlab. (2020). the.gitlab-ci.yaml file. Obtenido de https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html

Terraform. (2020). Deliver Infrastructure as Code. Obtenido de: https://www.terraform.io/

Puppet. (2020). Introduction to Puppet. Obtenido de: https://puppet.com/docs/puppet/6/puppet_overview.html

Redhat, (2017), Barclays adoptó la cultura ágil de DevOps para mantener la competitividad. Obtenido de RedHat success stories: https://www.redhat.com/es/success-stories/barclays

Yehuda, Y. (04 de mayo de 2017). How Barclays Does Comprehensive DevOps. Obtenido de Dbmaestro: https://www3.dbmaestro.com/blog/how-barclays-does-comprehensive-devops

Martin, A. (30 de mayo de 2018). Qúe es DevOps? Obtenido de Urtana: https://urtanta.com/que-es-devops/

Ambit-bts. (27 de noviembre de 2018). DevOps: Qúe es y su poder en la gestión de servicos de IT. Obtenido de Ambit: https://www.ambit-bst.com/blog/devops-que-es