TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

“Una solución para facilitar el despliegue y monitorización de microservicios en contenedores”

Curso 2018/2019

Alumno/a:

José Francisco Ruiz Zamora

Director/es:

Nicolás Padilla Soriano

Javier Criado Rodríguez

AGRADECIMIENTOS

A mis directores del trabajo, Nicolás Padilla y Javier Criado, por su ayuda para hacer posible este proyecto.

A mis padres y mi hermano por haber estado siempre conmigo.

Contenido

1. INTRODUCCIÓN ... 6
1.1. ARQUITECTURA ORIENTADA A SERVICIOS ... 6
1.2. MICROSERVICIOS ... 7
1.3. CONTENEDORES ... 9
1.3.1. AISLAMIENTO DE RECURSOS ... 9
1.3.2. DOCKER ... 10
1.4. DISTRIBUCIÓN DE SERVICIOS ... 11
1.4.1. KUBERNETES ... 11
1.4.2. ELECCIÓN DEL CLÚSTER ... 12
2. ANÁLISIS DEL SISTEMA ... 14
2.1. ANÁLISIS DAFO ... 14
2.2. FASES DE DESARROLLO ... 15
2.3. ESTIMACIÓN DE PLAZOS ... 17
3. CONTENEDORES: DOCKER ... 20
3.1. TECNOLOGÍA DE DOCKER ... 21
3.1.1. INSTALAR DOCKER EN UBUNTU ... 23
3.2 COMANDOS DE DOCKER ... 24
3.3. CONECTAR CONTENEDORES ... 27
3.4. DOCKERFILE ... 28
3.4.1. SINTAXIS DE UN DOCKERFILE ... 29
3.4.1.1. BUENOS HÁBITOS ... 31
3.5. ORQUESTACIÓN MULTICONTENEDOR ... 31
3.5.1. DOCKER COMPOSE ... 31
3.5.1.1. SINTAXIS ... 32
3.5.1.2. COMANDOS ... 33
4. ORQUESTACIÓN DE CONTENEDORES: KUBERNETES ... 36
4.1. OBJETOS DE KUBERNETES ... 36
4.1.1. POD ... 38
4.1.2. VOLUMEN ... 38
4.1.3. SERVICIO ... 39
4.1.4. NAMESPACE ... 39
4.1.5. REPLICASET ... 40
4.1.6. DEPLOYMENT ... 40
4.2. COMPONENTES DE KUBERNETES ... 40
4.3. EMPEZANDO CON KUBERNETES ... 43
4.3.1. INTERACTUAR CON EL CLÚSTER: KUBECTL ... 43
4.3.2. PREPARAR EL ENTORNO: MINIKUBE ... 44

4.4. ORQUESTACIÓN MULTICONTENEDOR: KOMPOSE ... 45
4.5. RED DE KUBERNETES ... 47
5. MONITORIZACIÓN ... 50
5.1. MONITORZACIÓN DE CONTENEDORES ... 50
5.2. MONITORIZACIÓN DEL CLÚSTER ... 53
5.2.1. KUBERNETES DASHBOARD ... 53
5.2.2. PROMETHEUS ... 56
5.2.2.1. DESPLEGAR PROMETHEUS EN EL CLÚSTER ... 57
6. ESCALADO DEL CLÚSTER ... 64
6.1. AUTO ESCALADOR: HPA ... 66
6.2. USO DE MÉTRICAS PERSONALIZADAS ... 68
6.3. PRUEBAS DE CARGA CON HEY ... 69
7. ESPECIFICACIÓN, DISEÑO Y DESARROLLO DE UNA INTERFAZ WEB ... 74
7.1. ANÁLISIS ... 74
7.1.1. ESPECIFICACIONES FUNCIONALES ... 74
7.2. DISEÑO ... 75
7.2.1. DIAGRAMA DE CASOS DE USO ... 75
7.2.2. PROTOTIPOS DE INTERFAZ ... 75
7.3. IMPLEMENTACIÓN ... 76
7.3.1. INICIAR LA APLICACIÓN EN UN SERVIDOR ... 77
7.3.2. DETALLES DE IMPLEMENTACIÓN ... 79
7.3.2.1. DESARROLLO INTERFAZ PRINCIPAL ... 81
7.3.2.2. DESARROLLO INTERFAZ DE RESULTADOS ... 83
7.4. PRUEBAS ... 85
8. ESCENARIO DE EJEMPLO ... 90
8.1. EJECUTANDO MICROSERVICIOS EXISTENTES ... 90
8.2. ENCAPSULACIÓN EN CONTENEDORES ... 91
8.2.1. UTILIZANDO DOCKER-COMPOSE ... 93
8.3. ORQUESTANDO EN KUBERNETES: MINIKUBE ... 94
8.3.1. ESCALANDO APLICACIONES: HPA ... 99
8.3.2. UTILIZANDO PROMETHEUS ... 101
8.4. INTERFAZ WEB ... 103
9. CONCLUSIONES ... 108
REFERENCIAS BIBLIOGRÁFICAS ... 112
ANEXO A: RECURSOS ADICIONALES ... 116

CAPÍTULO 1

INTRODUCCIÓN

1. INTRODUCCIÓN

El desarrollo de software utilizando una arquitectura basada en microservicios ha aumentado en los últimos años siendo adoptado por grandes empresas como Amazon, LinkedIn, Netflix o SoundCloud ya que propone notables mejoras frente a las arquitecturas monolíticas.

Uno de los factores del crecimiento de la arquitectura de microservicios se debe al reciente aumento en popularidad de las tecnologías basadas en contenedores y en orquestación de los mismos, lo que facilita la implementación y la hace más viable desde el punto de vista técnico.

Las tecnologías basadas en contenedores, como por ejemplo *Docker*, facilitan el despliegue de las aplicaciones. Mientras que para la orquestación de dichos contenedores existen distintos proyectos como *Docker Swarm*, o *Kubernetes*, siendo este último el referente actual.

El conjunto de las distintas tecnologías define el marco hacia donde se dirige el desarrollo de software tanto en la actualidad como en el futuro próximo. Es por tanto necesario conocer y entender no solo las nuevas tecnologías y herramientas, sino también la base sobre la que se encuentran.

En el contexto actual donde las visitas diarias a diferentes servicios webs se cuentan por millones se hace obligatorio optar por un modelo de desarrollo de aplicaciones distribuido y escalable y un despliegue automatizado.

Durante los últimos años las grandes empresas han desarrollado software y aumentado las funcionalidades propuestas al usuario haciendo uso de una arquitectura orientada a servicios. El crecimiento de dichas funcionalidades ha supuesto un problema ya que el acceso a ellas es un factor que no se puede controlar y no tiene porqué ser equitativo, por lo que una parte de la aplicación podría tener millones de peticiones y otra no llegar a unas pocas, además de aumentar el tamaño de las aplicaciones que pasaban a ser cada vez más pesadas. Y ya que un pequeño cambio requiere parar y volver a desplegar todo el conjunto, este modelo dificultaba al equipo de desarrollo el proceso de actualización o monitorización del sistema para encontrar un fallo.

Haciendo uso de una arquitectura de microservicios y herramientas como *Docker* y *Kubernetes* se puede lograr no solo el desarrollo y despliegue de nuevas funcionalidades en la aplicación final, sino también la monitorización para duplicar aquellos servicios que más peticiones reciban y repartir así la carga de manera automática o para controlar el ciclo de vida de cada uno de los servicios desplegados.

1.1. ARQUITECTURA ORIENTADA A SERVICIOS

La arquitectura orientada a servicios (*SOA* por sus siglas en inglés) es un patrón de arquitectura de software diseñado para satisfacer las necesidades comerciales de una organización en el que los componentes de la aplicación proporcionan servicios a otros componentes a través de la red mediante un protocolo de comunicación[1].

Hay dos roles definidos sobre los que se basa *SOA*, un consumidor (usuario humano, otro servicio o un proceso externo) que interactúa con la arquitectura y un productor, definido por el conjunto de servicios que forman la arquitectura y que ejecutan un proceso para llevar a cabo las peticiones del usuario, como muestra la Figura 1.

La orientación a servicios es una forma de pensar en términos de servicios, tanto en el desarrollo como en los resultados obtenidos, mientras que un servicio es una unidad de funcionalidad discreta a la que

[1] Chapter 1: Service Oriented Architecture (SOA). (2019). Recuperado de https://web.archive.org/web/20160206132542/https://msdn.microsoft.com/en-us/library/bb833022.aspx

se puede acceder de forma remota e interactuar (ejecutar o actualizar) de manera independiente. Se deben tener cuatro propiedades fundamentales para entrar dentro de la definición de *SOA*[2]:

1. Ser una representación lógica de una actividad repetitiva que tiene una salida específica.
2. Ser autocontenido, es decir, no requiere de otra actividad para que adquiera significado.
3. Ser una caja negra para sus consumidores.
4. Poder estar compuesto por otros servicios.

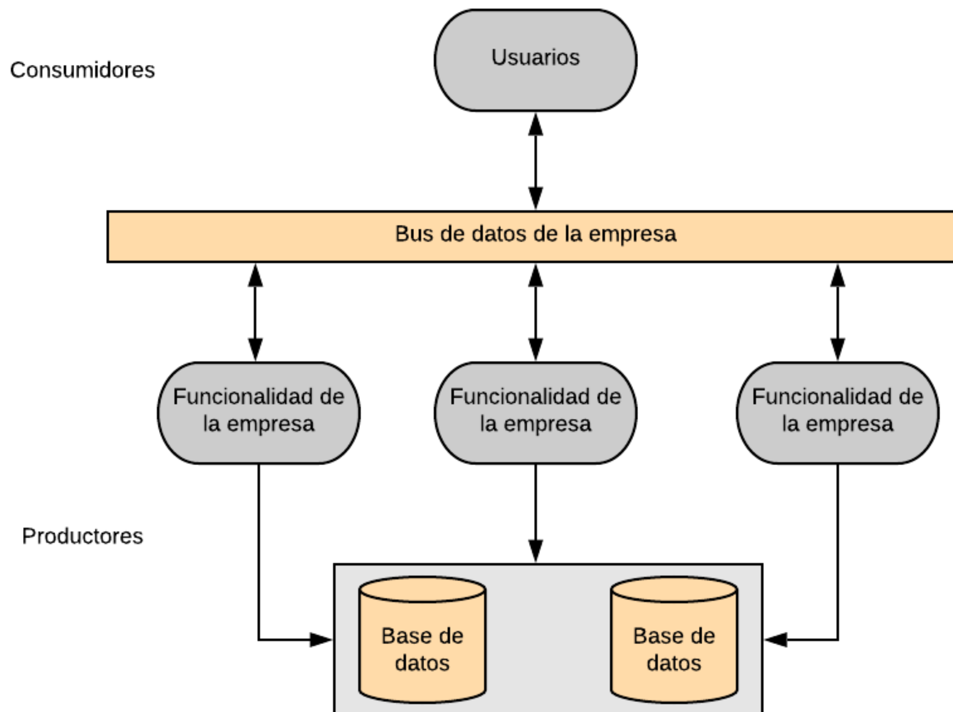

Figura 1: Arquitectura orientada a servicios

La arquitectura orientada a servicios es importante para distintos grupos de interés *(stakeholders)*, sobre todo desarrolladores y arquitectos ya que es una manera de crear aplicaciones dinámicas y colaborativas al permitir incorporar nuevas capacidades a lo largo del tiempo.

Un paso más allá de *SOA* para construir software distribuido es pensar en los servicios como procesos que interactúan a través de la red para cumplir un objetivo final utilizando protocolos independientes de la tecnología, encapsulando así el lenguaje y *frameworks*. Esto se conoce como microservicios.

1.2. MICROSERVICIOS

Se describe la arquitectura de microservicios como una forma de diseñar aplicaciones software como conjuntos de servicios implementados de manera independiente entre ellos, siendo esta arquitectura una aproximación para desarrollar una única aplicación como un conjunto de pequeños servicios, cada uno ejecutando su propio proceso y comunicándose entre ellos mediante mecanismos ligeros, generalmente una API basada en HTTP[3].

Aunque *SOA* y microservicios parezcan similares son realmente términos distintos ya que el primero es una arquitectura de más alto nivel ideada para diseñar aplicaciones basadas en servicios, pudiendo

[2] What Is SOA?. (2019). Recuperado de https://web.archive.org/web/20160819141303/http://opengroup.org/soa/source-book/soa/soa.htm
[3] Lewis, J., & Fowler, M. (2019). Microservices. Recuperado de https://martinfowler.com/articles/microservices.html

seguir estos servicios un diseño monolítico o de microservicios. La similitud viene dada ya que ambas son arquitecturas basadas en servicios, que generalmente son distribuidas, lo que significa que se puede acceder a los componentes de los servicios de forma remota[4].

La principal diferencia entre microservicios y la arquitectura monolítica es que ésta se construye como una única unidad por lo que cualquier cambio implicaría crear e implementar una nueva versión de la aplicación completa del lado del servidor. Toda la lógica para interpretar una solicitud se ejecuta en un único proceso y por tanto la única forma de dividir la aplicación es utilizando las herramientas del lenguaje en que se codifica como el uso de clases, funciones o *namespaces.*

Las aplicaciones monolíticas son utilizadas cada vez menos ya que un cambio realizado en una pequeña parte obliga a la reconstrucción y despliegue de toda la aplicación, lo que dificulta mantener un desarrollo limpio y una estructura modular con el paso del tiempo y los cambios. Además, el escalado de la aplicación no podría dividirse según la parte que solicite más recursos, sino que tendría que aplicarse en la aplicación completa.

Estos problemas han llevado a la creación y uso de la arquitectura de microservicios; crear aplicaciones como conjuntos de servicios, tal y como se muestra en la Figura 2[5]. Al ser los servicios desplegables y escalables de forma independiente pueden ser ejecutados incluso en diferentes ordenadores y no tienen que compartir el mismo lenguaje de programación.

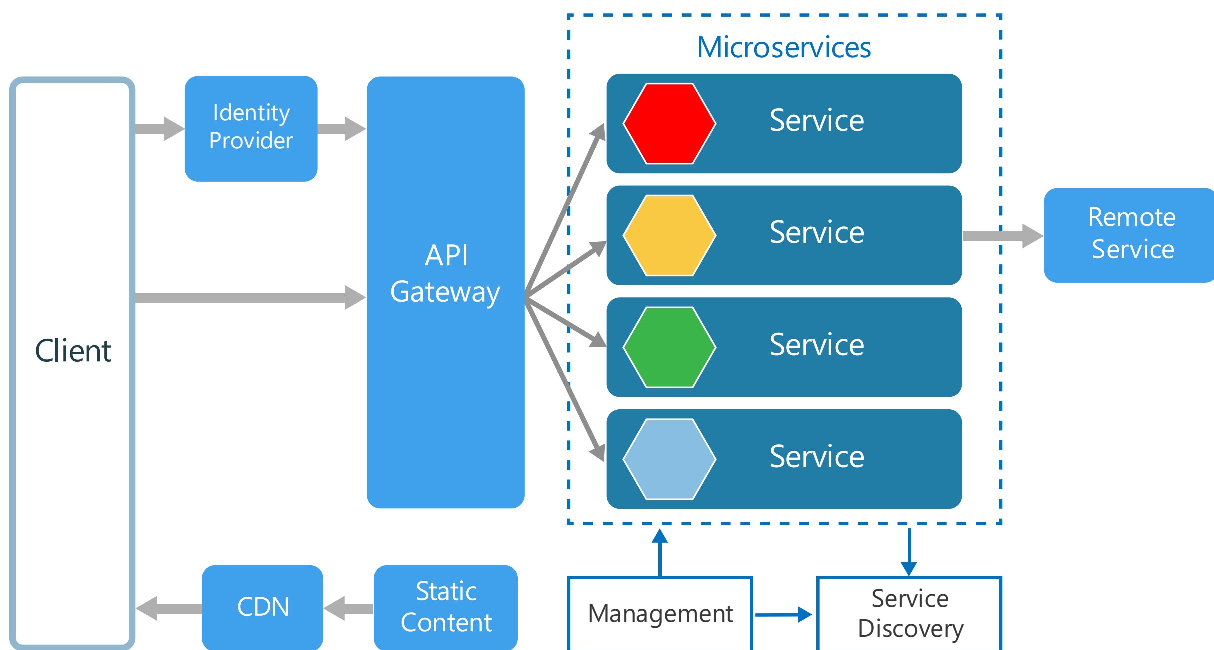

Figura 2: Arquitectura de Microservicios

Junto a la idea de deshacerse de una arquitectura cada vez más obsoleta para grandes aplicaciones, como es la monolítica, el auge de los microservicios tiene sus raíces en el aumento de tecnologías basadas en contenedores.

[4] Microservices vs. SOA - DZone Microservices. (2019). Recuperado de https://dzone.com/articles/microservices-vs-soa-2

[5] Microsoft - Microservices architecture style. (2018). Recuperado de https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

1.3. CONTENEDORES

Los contenedores son paquetes que se basan en el aislamiento virtual para implementar y ejecutar aplicaciones que acceden a un *kernel* de sistema operativo compartido sin la necesidad de máquinas virtuales[6].

Las tecnologías basadas en contenedores nacen del concepto de particionamiento o segmentación del hardware y del aislamiento del proceso ***chroot***. Aunque el término contenedor es relativamente nuevo los sistemas operativos UNIX usaban "jaulas" como forma de aislar un proceso y sus subprocesos del resto del sistema para evitar acceder a recursos protegidos. El objetivo final de los contenedores ha crecido respecto a las jaulas queriendo no prohibir el acceso a determinados recursos sino aislar a un proceso de todos los recursos excepto a los que se le hayan permitido explícitamente para su ejecución.

El uso de contenedores es una buena práctica para el desarrollo de software, pero la construcción manual de los mismos puede ser una tarea complicada que da lugar a errores. Con la intención de solucionar este problema nació Docker; así cualquier programa que se ejecute con Docker se está ejecutando en un contenedor consistente construido según los mejores métodos de desarrollo[7].

1.3.1. AISLAMIENTO DE RECURSOS

Poder mantener los recursos aislados es la idea básica de los contenedores. Cuando una aplicación es iniciada, inevitablemente se utilizan recursos compartidos por el resto de programas que se encuentran en el host como CPU, espacio de memoria o lectura y escritura en disco.

Existen distintos métodos de aislamiento, siendo el más popular la creación de Máquinas Virtuales para disponer de un entorno seguro y consistente. La implementación de una máquina virtual se realiza virtualizando la capa hardware mediante un hipervisor, donde para ejecutar una aplicación primero se debe instalar un sistema operativo completo. Es decir, los recursos se aíslan entre los sistemas operativos que comparten el hipervisor. Mientras que el contenedor está construido sobre el sistema operativo que lo ejecuta utilizando únicamente los recursos necesarios para su funcionamiento.

Como se muestra en la Figura 3, el contenedor aísla una aplicación en la capa del Sistema Operativo (OS) por lo que no necesita uno propio como en el caso de las máquinas virtuales, lo que hace a los contenedores más ligeros y portables.

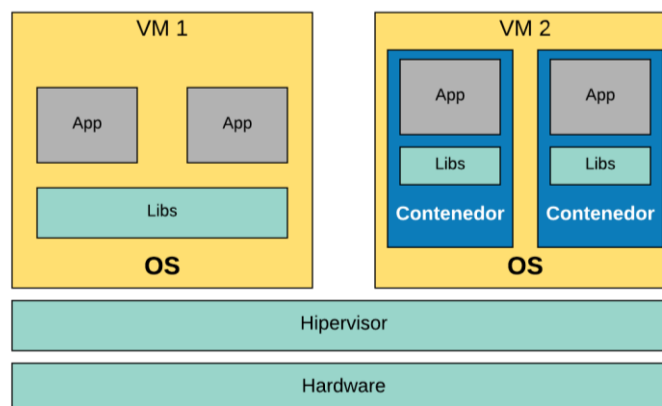

Figura 3: Arquitectura de una máquina virtual y un contenedor

[6] What is container (containerization or container-based virtualization)? - Definition from WhatIs.com. (2019). Recuperado de https://searchitoperations.techtarget.com/definition/container-containerization-or-container-based-virtualization

[7] Nickoloff, J., (2016). *Docker in Action*, Shelter Island, NY: Manning Publications Co.

1.3.2. DOCKER

Docker es una plataforma de código abierto para desarrollar, montar y ejecutar aplicaciones que permite separar dichas aplicaciones de la infraestructura del ordenador, por lo que se mejoran tiempos de entrega o pruebas del software[8].

La herramienta Docker ofrece la capacidad de empaquetar y ejecutar una aplicación en un entorno aislado llamado contenedor. Como se explica anteriormente, los contenedores se ejecutan en el *kernel* del host, por lo que al no ser necesaria la carga adicional de un hipervisor o monitor de máquina virtual es posible ejecutar una carga mayor de contenedores en una combinación de hardware que si se estuviesen usando máquinas virtuales.

Docker basa su funcionamiento en una arquitectura cliente-servidor con los componentes principales definidos por:

- Un servidor donde se ejecuta el llamado demonio (*daemon*) que se encarga de gestionar los objetos de Docker según las peticiones que recibe.
- Una API REST utilizada por los programas a modo de interfaz para comunicarse con el *daemon*.
- Una interfaz de línea de comandos (CLI) para la parte del cliente.

La lógica de funcionamiento es que la interfaz de línea de comandos, según las peticiones que reciba del cliente, utiliza la API REST para comunicarse con el *daemon*. Es el propio *daemon* de Docker quién crea y gestiona los objetos como imágenes, contenedores, red y volúmenes.

En la Figura 4[9] se observan los distintos niveles de Docker siendo el núcleo la parte servidor que controla todo el programa.

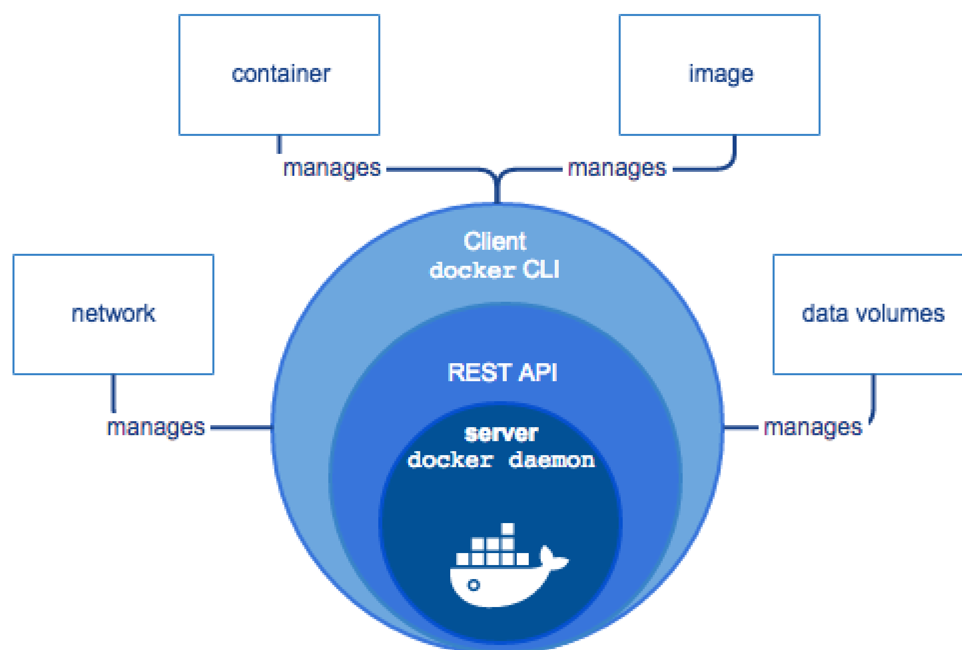

Figura 4: Arquitectura de Docker

[8] Docker overview. (2019). Recuperado de https://docs.docker.com/engine/docker-overview/
[9] Docker Engine (2019). Arquitectura de Docker. Recuperado de https://docs.docker.com/engine/docker-overview/

1.4. DISTRIBUCIÓN DE SERVICIOS

Los beneficios que proporcionan los contenedores hacen que no se utilicen únicamente para encapsular aplicaciones simples o algunas de sus dependencias. Son muchas las grandes empresas que confían en *Docker* y lo incorporan a nivel de negocio. Con la irrupción de dichas empresas y sus aplicaciones de gran tamaño con múltiples dependencias y enlaces se hace imposible manejar todos los contenedores de forma individual, lo que da lugar a la necesidad de lo que se conoce como un sistema de orquestación.

Se conoce como orquestación en el ámbito de la computación a la configuración, coordinación y gestión de sistemas informáticos y software[10] y su uso se debe a múltiples factores como la necesidad de escalar servicios según sus necesidades, construir servicios a través de múltiples máquinas para compartir recursos o administrar de manera simple una arquitectura completa de microservicios, además un sistema de orquestación ayuda a los usuarios a ver un conjunto de host (también conocidos como nodos) como un clúster seguro que puede tratarse como si fuese un ordenador único a gran escala. *Kubernetes* es un programa que permite gestionar la orquestación de contenedores de forma sencilla y, a día de hoy, es el más popular entre los usuarios[11].

Hoy en día cada vez más servicios conectan tanto entre ellos como por la red mediante APIs que generalmente se encuentran en sistemas distribuidos, es decir, las distintas partes que la componen se ejecutan en distintas máquinas conectándose a través de la red. Debido a la dependencia que se tienen de estas APIs su uso es significativamente alto y es por ello que los sistemas deben mantener tanto la disponibilidad en todo momento incluso durante la implementación de actualizaciones o las tareas de mantenimiento como la escalabilidad para mantenerse en funcionamiento con cada día mayor carga de trabajo sin necesidad de modificar la estructura y el diseño.

1.4.1. KUBERNETES

Conocido como *K8s*, *Kubernetes* es un sistema de código abierto desarrollado originalmente por Google y utilizado para automatizar el despliegue, escalado y administración de aplicaciones ejecutadas en contenedores a través de múltiples hosts[12]. Al igual que los contenedores, *Kubernetes* está diseñado para poder ejecutarse en cualquier lugar desde cero, ya sea un ordenador local, la nube pública o híbrida.

El uso de *Kubernetes* no está únicamente orientado a las necesidades de grandes compañías sino también para proyectos a menor escala o para desarrolladores que quieran crear contenido propio fuera del mercado. Desde un clúster de Raspberry Pi hasta uno formado por los últimos modelos de ordenadores existentes en el mercado, *Kubernetes* proporciona el software necesario para construir, implementar y configurar sistemas distribuidos confiables y escalables. Ya que contiene las necesidades más importantes para ejecutar aplicaciones en contenedores como son:

- Despliegue de contenedores
- Almacenamiento persistente
- Monitorización del estado de los contenedores
- Gestión de recursos
- Escalado automático
- Robustez del clúster

[10] Erl, T. (2004). *Service-Oriented Architecture*. East Rutherford, NJ: Prentice Hall PTR.
[11] Goasguen, S. (2015). *Docker Cookbook.* Sebastopol, CA: O'Reilly Media, Inc.
[12] Kubernetes Documentation. (2019). Recuperado de https://kubernetes.io/docs/home/

Kubernetes sigue una arquitectura maestro-esclavo donde cada rol tiene tareas distintas, el maestro controla y programa todas las actividades del clúster mientras que los trabajadores son nodos donde se ejecutan los contenedores como se muestra en la Figura 5[13].

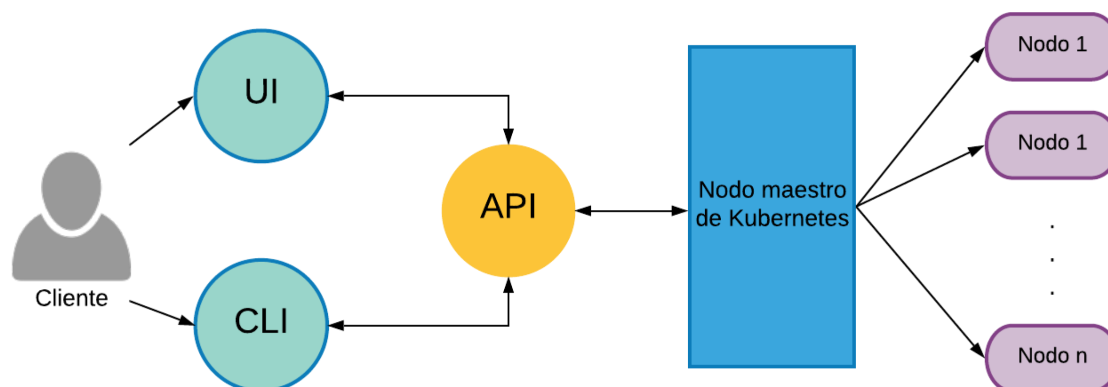

Figura 5: Arquitectura de Kubernetes

La facilidad de disponer de múltiples contenedores orquestados convierte a *Kubernetes* en un complemento perfecto para los microservicios.

1.4.2. ELECCIÓN DEL CLÚSTER

Como se menciona anteriormente el clúster de *Kubernetes* puede ejecutarse en distintas plataformas, tanto a nivel local como en la nube. Para cada opción hay distintas posibilidades donde el esfuerzo requerido varía desde ejecutar un único comando a tener que preparar el clúster personalizado. La elección final donde ejecutar *Kubernetes* varía de las necesidades de cada proyecto por lo que no existe una mejor o peor solución, aunque sí es cierto que una solución en la nube facilita la creación y el mantenimiento ya que está a cargo del servidor donde se aloje, generalmente grandes compañías[14].

Las soluciones más apoyadas por la comunidad de *Kubernetes* para crear un clúster local son:

- *Minikube*: Método para crear un clúster de un único nodo (siendo el mismo nodo maestro y esclavo) ideal para el desarrollo y pruebas. La instalación es automatizada y no se requiere de un proveedor en la nube.
- *Kubeadm*: Clúster de varios nodos (maestros y esclavos) que solo requiere el uso del motor de Docker, es decir, solo requiere que esté instalado Docker.

Mientras que para los servicios en la nube la popularidad pertenece a grandes empresas como Google, Microsoft, IBM o Amazon, siendo las soluciones más utilizadas:

- Amazon Elastic Container Service for Kubernetes
- Azure Kubernetes Service
- DigitalOcean Kubernetes
- Google Kubernetes Engine
- IBM Cloud Kubernetes Service

---

[13] Saito, H., Lee, H., & Wu, C. (2017). *DevOps with Kubernetes*. Birmingham, Reino Unido: Packt Publishing Ltd.
[14] Picking the Right Solution. (2019). Recuperado de https://v1-12.docs.kubernetes.io/docs/setup/pick-right-solution/

CAPÍTULO 2

ANÁLISIS DEL SISTEMA

2. ANÁLISIS DEL SISTEMA

A continuación, se muestra mediante un análisis DAFO todas las variantes y aspectos del proyecto realizado, además de plantear las distintas fases y los plazos estimados para su elaboración. El análisis DAFO es un proceso en el que el equipo de un proyecto identifica los factores internos y externos que afectarán al futuro del mismo. Las **fortalezas** y **debilidades** son los factores internos mientras que las **oportunidades** y **amenazas** son los externos.

2.1. ANÁLISIS DAFO

Antes de comenzar con el desarrollo del proyecto es conveniente realizar un análisis DAFO para planificarlo de mejor manera asignando eficientemente los recursos disponibles. Cada factor estudia un motivo específico; el estudio de las **debilidades** se utiliza, no como una crítica hacia el proyecto, sino para conocer las áreas más vulnerables que deben mejorarse, las **oportunidades** ayudan a la planificación a largo plazo, las **amenazas** a estar preparado ante una adversidad y evitar la menor pérdida posible mientras que las **fortalezas** permiten tener una idea sobre como posicionarse respecto al producto para obtener el mayor valor posible.

FORTALEZAS:

- No es necesario un equipo con grandes características donde desarrollar y ejecutar el proyecto.
- Las herramientas utilizadas son de código abierto.
- Las herramientas utilizadas tienen la confianza de la comunidad y por tanto aceptación para utilizarlo en sus aplicaciones.

DEBILIDADES:

- Necesidad de estudio y especialización en las tecnologías y herramientas utilizadas.
- La herramienta ofrecida no se encuentra en el mercado, pero la combinación de otras herramientas podría sustituirla.

OPORTUNIDADES:

- Pocas alternativas en el mercado.
- A partir del proyecto actual sería posible crecer para combinar la funcionalidad de distintas herramientas en una única.
- El auge de las herramientas hace que sea un proyecto con gran proyección de cara al futuro.

AMENAZAS:

- Herramientas relativamente nuevas, lo que puede suponer grandes cambios en poco tiempo según las actualizaciones.
- El proyecto podría quedarse obsoleto debido al enorme crecimiento de las herramientas utilizadas.

2.2. FASES DE DESARROLLO

Las fases de desarrollo en las que se ha dividido el trabajo a realizar son las siguientes:

1. Conocimiento sobre el proyecto

Esta primera fase consiste en conocer el proyecto sobre el que se quiere trabajar, cuáles son las metas y objetivos, las tecnologías y herramientas necesarias y los posibles problemas que habrá que afrontar. En resumen, tener una idea global del proyecto a realizar. Las etapas de esta fase son:

- Descripción del proyecto.
  Conocer el problema a resolver y enfocar la solución posible con las herramientas disponibles hoy en día, así como saber otros factores como las motivaciones para la realización del mismo o la extensión prevista.
- Estudio del mercado.
  Valorar las opciones disponibles en el mercado, tanto de herramientas útiles para la tecnología del proyecto como aplicaciones similares. En este punto se deciden las tecnologías finales con las que desarrollar el proyecto, siendo las elegidas para encapsulación en contenedores y para orquestación, *Docker* y *Kubernetes* respectivamente.
- Definición de los objetivos.
  Conocer la finalidad del proyecto y los objetivos que debe cubrir para considerar que es funcional. Después de este punto se sabe que se quiere un sistema que "automatice el despliegue orquestado de contenedores y recoja métricas para escalar aquellos más solicitados según peticiones HTTP enviadas. Además, el proceso de envío de peticiones y comportamiento del clúster debe ser accesible por el usuario mediante una interfaz web".
- Identificación de los componentes necesarios para realizar la solución.
  Tener conocimiento de los recursos hardware y software que van a ser necesarios durante el desarrollo del proyecto, que en este caso son:
    - Hardware
        - Equipo local o remoto donde ejecutar el clúster y la aplicación web.
    - Software
        - Herramienta para encapsular en contenedores.
        - Herramienta para orquestar contenedores.
        - Herramienta para monitorizar el clúster.
        - Plataforma para la que desarrollar la aplicación final.

2. Estudio de las tecnologías

Una vez conocidos todos los detalles referentes al proyecto, la siguiente fase consiste en el estudio y conocimiento de las herramientas y tecnologías a utilizar. Las nuevas tecnologías son microservicios, contenedores y la orquestación de los mismos. Para ello es necesario llevar a cabo:

- Aprendizaje sobre microservicios: Conocimiento de la arquitectura basada en microservicios, en qué consiste y con qué se trabaja en el proyecto, además del uso de *Spring Boot* para facilitar el despliegue.
- Aprendizaje sobre contenedores: Entender el concepto teórico, la funcionalidad y el uso correcto de los contenedores en el ámbito del desarrollo, así como la herramienta *Docker* para llevarlo a cabo.
- Aprendizaje sobre orquestación: Conocer el funcionamiento y la lógica de la orquestación de contenedores, los eventos que ocurren en el *backend* del clúster, así como el uso de *Kubernetes*, la herramienta elegida para el desarrollo del proyecto.

3. Levantamiento del clúster

La siguiente fase consiste en la puesta en funcionamiento del clúster de contenedores aplicando todos los conocimientos aprendidos anteriormente sobre los microservicios reales en los que se van a trabajar cumpliendo las siguientes etapas:

- Despliegue de microservicios.
  Lanzar los microservicios en un entorno libre (no en contenedores) de manera que se pueda comprobar su correcto funcionamiento.
- Encapsulación de los microservicios en contenedores
  Mediante *Docker* aislar los microservicios en contenedores y realizar toda la configuración necesaria para que sean funcionales.
- Orquestación de los contenedores en un clúster.
  Configurar un clúster mediante *Kubernetes* en el que se ejecuten los contenedores levantados previamente.
- Monitorización del clúster.
  Estudiar los eventos que ocurren en el clúster una vez que está levantado tales como métricas, *logs*, errores o cualquier otra información que proporcionen los contenedores y pueda ser utilizada en el desarrollo de la interfaz web.

4. Métricas personalizadas en el clúster

Accediendo a los datos del clúster una vez que se ejecuta correctamente el siguiente paso es hacer uso de métricas personalizadas para el escalado de las aplicaciones, ya que por defecto no se pueden medir las peticiones HTTP que recibe un servicio del clúster. Será necesario la configuración y despliegue de herramientas que permitan a *Kubernetes* acceder a métricas que no proporciona su propia API. Para ello se hará uso de un software externo, *Prometheus*. Las etapas que se deben superar son:

- Instalación y configuración del software externo, *Prometheus*.
  Para la lectura por parte de *Kubernetes* de métricas distintas al uso de memoria o CPU es necesario el uso de métricas personalizadas. La configuración del clúster para su lectura no es sencilla, es por eso que se hará uso de *Prometheus.*
- Configuración API *Kubernetes.*
  Ya que por defecto *Kubernetes* no soporta ciertas métricas, éstas no serán accesibles mediante su API, es por eso que es necesario configurarla.
- Pruebas de carga.
  Una vez realizada la configuración aplicar carga de trabajo a los servicios dentro del clúster y comprobar el comportamiento.

5. Desarrollo aplicación web

En esta fase se desarrolla por completo la interfaz web de la aplicación que posteriormente usará el usuario.

- Análisis.
  Estudiar los requisitos necesarios que debe cumplir la aplicación así como la manera en que se desarrollará, tanto lenguaje como técnicas a utilizar.
- Diseño.
  Conociendo el marco en el que queda definida la aplicación en cuanto a requisitos y funcionalidad se diseña la interfaz mediante la cual el usuario interactúa con la aplicación.
- Codificación.

En la etapa de codificación se transcriben las ideas propuestas anteriormente a un código funcional.

- Pruebas
  Por último, queda probar el correcto funcionamiento de la aplicación mediante pruebas que verifiquen que los valores de salida que se muestran se corresponden con los valores reales que tiene *Kubernetes*.

2.3. ESTIMACIÓN DE PLAZOS

Con todo lo explicado anteriormente se muestra a continuación, en la Tabla 1, el cronograma correspondiente a la duración de las diferentes fases del proyecto, además del diagrama de Gantt resultante en la Figura 6. Se incluyen las fechas de inicio, fin y tiempo estimado sin contar fines de semana como días de trabajo para cada tarea, siendo un proceso secuencial al tratarse de un trabajo unipersonal.

	DURACIÓN(DÍAS)	FECHA INICIO	FECHA FIN
CONOCIMIENTO SOBRE EL PROYECTO	**6**	**01/10/18**	**08/10/18**
ESTUDIO DE LAS TECNOLOGÍAS	**40**	**09/10/18**	**03/12/18**
APRENDIZAJE SOBRE MICROSERVICIOS	11	9/10/18	23/10/18
APRENDIZAJE SOBRE DOCKER	16	24/10/18	14/11/18
APRENDIZAJE SOBRE KUBERNETES	13	15/11/18	03/12/18
LEVANTAMIENTO DEL CLÚSTER	**35**	**04/12/18**	**01/04/19**
DESPLIEGUE DE MICROSERVICIOS	4	04/12/18	07/12/18
ENCAPSULACIÓN DE MICROSERVICIOS	11	10/12/18	24/12/18
ORQUESTACIÓN DE CONTENEDORES	20	25/12/18	21/01/19
USO DE MÉTRICAS PERSONALIZADAS	**34**	**13/02/19**	**01/04/19**
CONFIGURACIÓN DE PROMETHEUS	8	13/02/19	22/02/19
CONFIGURACIÓN API DE KUBERNETES	20	25/02/19	22/03/19
PRUEBAS DE CARGA	6	25/03/19	01/04/19
DESARROLLO APLICACIÓN WEB	**31**	**15/04/19**	**27/05/19**
ANÁLISIS	3	15/04/19	17/04/19
DISEÑO	2	18/04/19	19/04/19
CODIFICACIÓN	20	22/04/19	17/05/19
PRUEBAS	6	20/05/19	27/05/19
TOTAL DURACIÓN (DÍAS)	**146**	**01/10/18**	**27/05/19**

Tabla 1: Cronograma con las diferentes fases del proyecto

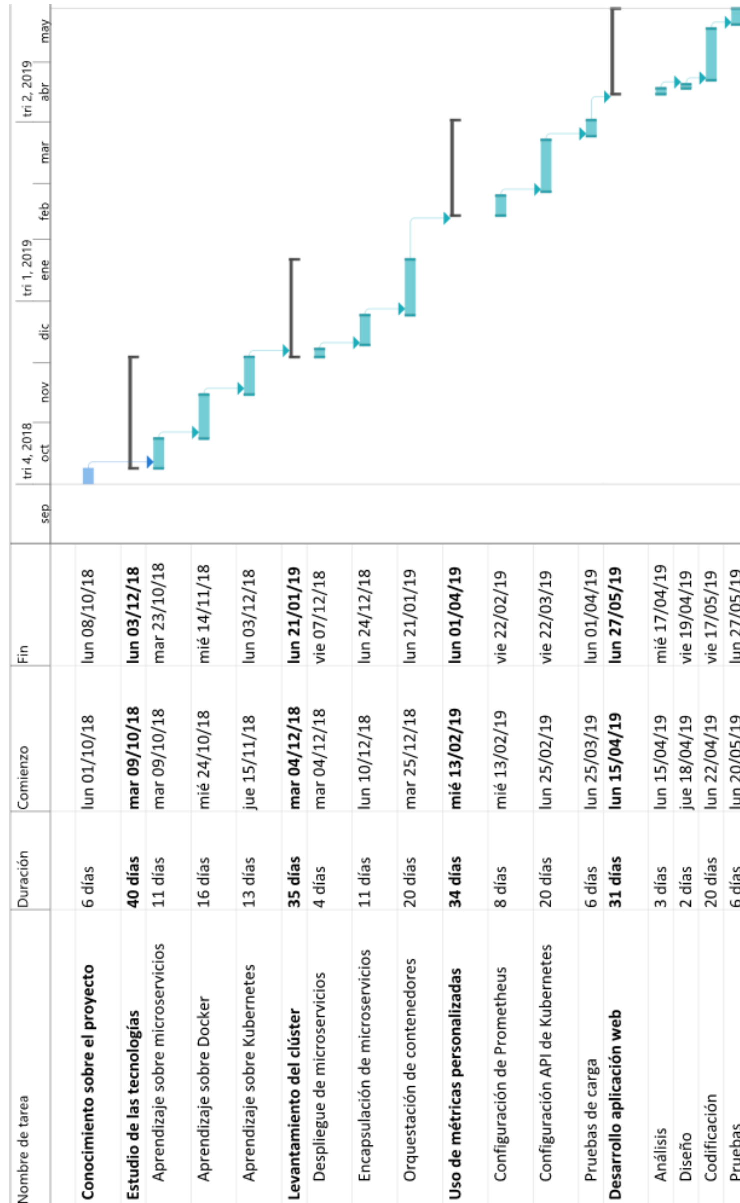

Figura 6: Diagrama de Gantt

CAPÍTULO 3

CONTENEDORES: DOCKER

3. CONTENEDORES: DOCKER

La herramienta *Docker* permite encapsular aplicaciones en contenedores y resolver problemas para toda una organización, desde facilitar el trabajo a los desarrolladores hasta complementar la funcionalidad de la propia empresa.

Para los desarrolladores el principal problema que se encuentran a la hora de crear software es la contaminación del equipo de trabajo, ya que incluso teniendo el máximo cuidado durante el desarrollo este problema es una realidad a tener en cuenta. Ocurre cuando el equipo donde se está desarrollando el software tiene una configuración que impide el correcto funcionamiento o que crea una falsa idea de una ejecución satisfactoria, siendo el ejemplo más claro el uso de rutas absolutas para guardar imágenes, que en local hará que se vean correctamente, pero al ejecutar el programa en un servidor no encontrará las imágenes a mostrar. También se puede dar el problema con el uso de distintas versiones de ciertos elementos. En un mundo ideal tanto el entorno de producción como el de desarrollo deberían ser coherentes pero la realidad es distinta y es por ello que surge este problema.

Con *Docker* se soluciona de manera sencilla ya que se puede crear un entorno aislado que contenga todos los recursos que el desarrollador elija de forma fácil, por lo que cualquier proyecto ejecutado dentro del contenedor podría equipararse a estar ejecutado en un servidor remoto. Además, los contenedores son fáciles de crear, eliminar y/o reiniciar, lo que facilita las pruebas de software.

Dentro del ámbito de una empresa cualquier error o pérdida de tiempo en la aplicación significa pérdida de dinero y reputación, por lo que las pruebas de software deben ser cada vez más exhaustivas lo que conlleva que las nuevas funcionalidades quedan paradas en segundo plano mientras que se configura el entorno de pruebas, se ejecuta la aplicación en dicho entorno, se corrigen los fallos hasta que se pasen los test y se valoran los cambios necesarios para implementar la aplicación actualizada en producción. Es decir, una nueva versión de la aplicación podría esperar desde días a meses, dependiendo de la complejidad y el riesgo de los cambios a introducir, para estar en funcionamiento, lo cual supone un problema a nivel de negocio.

Utilizando *Docker* no se puede acortar o saltar el proceso descrito para implementar una mejora ya que son pasos necesarios para tener un software de garantías, pero sí permite facilitar y agilizar el proceso al ser un entorno consistente donde los desarrolladores trabajan con la misma configuración del contenedor que se encuentra en producción. Por ejemplo, cuando un desarrollador compruebe el correcto funcionamiento del código en local también puede saber que se ejecutará correctamente en otro entorno, como se explica anteriormente, por lo que puede iniciar las pruebas de software en el mismo momento en el que termine las modificaciones, sin pasos intermedios. Cada vez que se realice un paquete de pruebas se elimina el contenedor y se realizan las siguientes pruebas, lo que hace que la parte de *testing* sea mucho más flexible y pueda reutilizar un mismo entorno en lugar de redistribuir o reiniciar los servidores para cada paquete de pruebas. Gracias a esta simplificación del proceso los cambios realizados en la aplicación llegan antes a producción lo que permite lanzar nuevas funcionalidades en menos tiempo.

También es beneficioso a nivel empresarial protegiendo a los ordenadores de las aplicaciones que se ejecuten en su interior. Un programa puede dañar un ordenador de múltiples maneras, desde ser un programa malicioso con intención directa de atacar la máquina hasta errores de programación que pueden o bien dañar el equipo o dejar abierta una puerta trasera.

Sea cual sea la finalidad, la seguridad del equipo queda expuesta. Con el uso de contenedores los programas que se ejecuten pueden acceder únicamente a los recursos disponibles dentro del mismo (Figura 7), aunque de forma excepcional previo permiso explícito del usuario, se pueden acceder a

recursos fuera del contenedor, todos los demás recursos entre los que se puede incluir información sensible de la empresa no corren peligro.

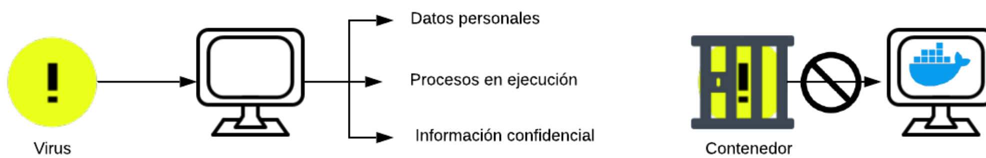

Figura 7: Seguridad de los contenedores

3.1. TECNOLOGÍA DE DOCKER

El núcleo de *Docker* está formado por el motor (*daemon*) que permite crear y administrar los contendores, es por eso que para empezar a utilizar la herramienta es necesario instalar dicho motor en un host, ya sea ordenador de escritorio, portátil o un servidor remoto.

Docker utiliza una arquitectura cliente-servidor donde la parte del cliente se comunica con el *daemon* para que éste se encargue de construir, ejecutar y distribuir los contenedores, pudiendo ejecutarse ambas partes tanto en el mismo host como en plataformas distintas ya que la comunicación entre ellas se realiza mediante una API REST a través de *sockets* UNIX o una interfaz de red [15].

Como se muestra en la Figura 8, las peticiones que realiza el cliente a través del CLI se envían al motor (servidor) para realizar las acciones necesarias.

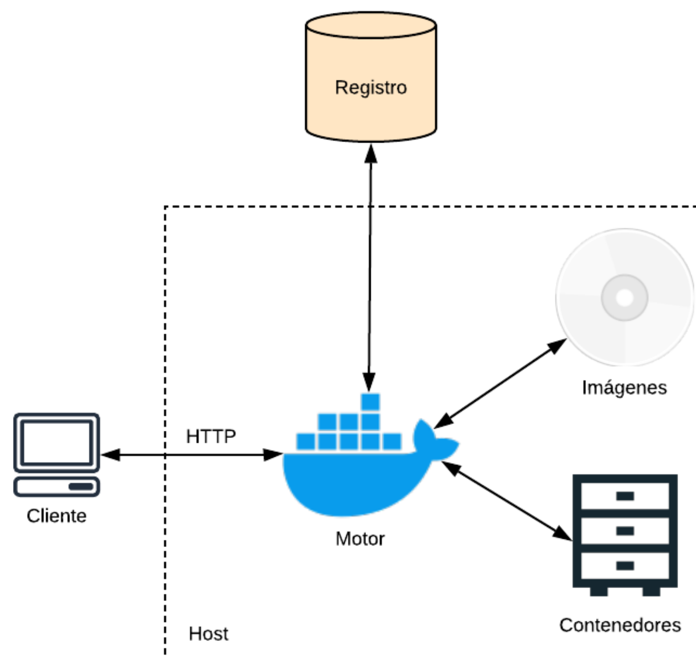

Figura 8: Arquitectura de Docker

El motor (*daemon*) de *Docker* se encarga de escuchar las solicitudes de la API tanto de un cliente como de otro motor y administrar los objetos propios de *Docker* como imágenes, contenedores, redes o volúmenes. Mientras que en el lado del cliente, el CLI es la manera principal para interactuar con

[15] Docker overview. (2019). Recuperado de https://docs.docker.com/engine/docker-overview/

Docker enviando los comandos al *daemon* mediante la API para que se ejecuten, pudiendo el propio cliente interactuar con más de un motor.

Por otro lado, se encuentra el registro, donde se almacenan las imágenes de *Docker*. *Docker Hub* es un registro público y es el lugar donde *Docker* está configurado para buscar por defecto al ejecutar los comandos *Docker pull* (descargar imagen) o *Docker run* (ejecutar un contenedor) y para insertar una imagen personalizada mediante el comando *push* (subir imagen al registro). Estos y más comandos se explicarán con más detalle más adelante.

Al levantar un contenedor se crean distintos objetos, instancias o procesos, siendo los más significativos de cara al usuario los propios contenedores y las imágenes que ejecutan en su interior, aunque no son los únicos.

Una **imagen** es una plantilla de lectura que contienen instrucciones a partir de las cuales se crean los contenedores. Cada imagen creada puede contener a su vez otra imagen junto con recursos extra personalizados, como por ejemplo, se puede crear una imagen personalizada basada en la ya existente de *Ubuntu* añadiendo distintos comandos para instalar o ejecutar algún programa o configuración necesaria como podría ser un servidor Apache.

Se pueden crear imágenes propias o utilizar ya existentes publicadas en el registro de *Docker*. Generalmente las organizaciones hacen disponible la imagen necesaria para levantar los contenedores y ejecutar sus aplicaciones de manera automática lo que facilita el trabajo a la hora de crear servidores o bases de datos.

Una imagen propia se crea a través de un documento llamado *Dockerfile* con una sintaxis concreta que define los pasos necesarios para la ejecución. En el proceso de la construcción de la imagen se crea una capa por cada instrucción declarada en el *Dockerfile*, así cuando éste cambia, en lugar de reconstruir la imagen completa, únicamente se reconstruyen las capas que han cambiado haciendo por tanto que las imágenes no sean tan pesadas en comparación con otras tecnologías de virtualización. El proceso se define en la Figura 9.

Un **contenedor** es una instancia ejecutable de una imagen que se puede crear, iniciar, detener, mover o eliminar. Un contenedor puede ser conectado a una o más redes, adjuntarle un espacio de memoria para almacenamiento persistente o incluso crear una nueva imagen según el estado actual, siendo esta última opción una gran utilidad para el escalado.

Los contenedores están definidos por las imágenes mediante las cuales han sido creados, así como por las opciones de configuración que se les puede proporcionar mediante el CLI en determinados momentos como en la creación. Al eliminar el contenedor se eliminan todos las modificaciones que se hayan hecho sobre él, siempre que no se almacenen en un espacio de memoria persistente.

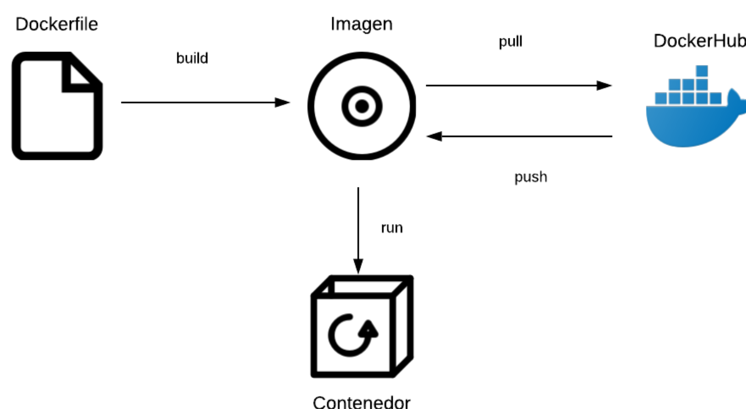

Figura 9: Imágenes en Docker

3.1.1. INSTALAR DOCKER EN UBUNTU

Conocida la forma en que trabaja *Docker* para poder ejecutarlo en un ordenador con Ubuntu es necesario instalarlo. En Ubuntu se puede realizar haciendo uso de un único comando en la Figura 10:

```
$ apt install docker.io
```

Figura 10: Instalación de Docker

Para comprobar la correcta instalación se puede ejecutar un primer contenedor de prueba llamado *hello-world* que trae por defecto con la instalación, para ello únicamente es necesario escribir en la línea de comandos: `$ sudo docker run hello-world`

La salida mostrada por pantalla debe ser similar a la mostrada en la Figura 11.

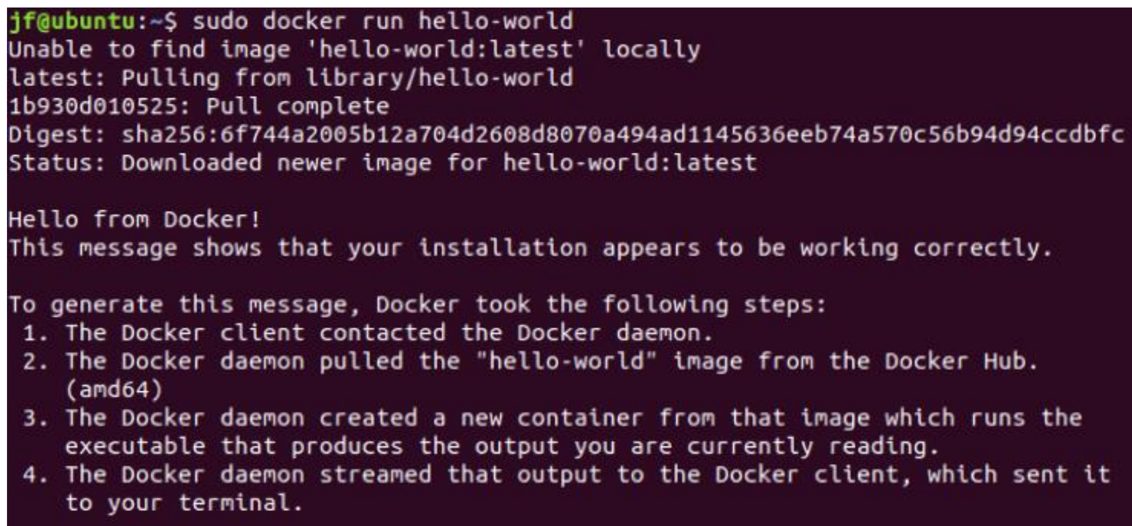

Figura 11: Ejecución de Docker

El contenedor indica la correcta instalación y el buen funcionamiento del mismo, y por tanto de *Docker,* además de los pasos que ejecuta internamente para el levantamiento del contenedor.

Acabamos de ejecutar el comando `run` para poner en marcha un contenedor. Esta palabra reservada es uno de los comandos más importantes, pero no el único. La totalidad de comandos se puede observar mediante el parámetro `--help`.

En la Figura 12 se muestran unos pocos comandos. En la documentación oficial, (disponible en https://docs.docker.com/engine/reference/commandline/cli/) se pueden encontrar la totalidad de ellos además de una explicación para cada uno.

```
jf@ubuntu:~$ docker --help

Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:
      --config string      Location of client config files (default
                           "/home/jf/.docker")
  -D, --debug              Enable debug mode
  -H, --host list          Daemon socket(s) to connect to
  -l, --log-level string   Set the logging level
                           ("debug"|"info"|"warn"|"error"|"fatal")
                           (default "info")
```

Figura 12: Comandos de Docker

3.2 COMANDOS DE DOCKER

La manera que tiene el usuario de interactuar con el motor de *Docker* es mediante comandos introducidos por el CLI. Estos comandos permiten trabajar de forma sencilla y simple con contenedores ejecutando acciones o modificando la configuración de los mismos tanto si están en ejecución como si no.

Cabe destacar que, al igual que en el entorno UNIX existen *flags* que no necesitan un valor como pueden ser `-l` o `-ls`, también existen en *Docker* y se denominan "*boolean flags*" ya que están indicados o no lo están. Para conocer los indicadores existentes para cada comando es necesario hacer uso del comando `--help` y así acceder a la ayuda de *Docker*.

Por ejemplo, mediante el comando `docker images --help` se tiene la respuesta mostrada en la Figura 13:

```
jf@ubuntu:~$ docker images --help

Usage:  docker images [OPTIONS] [REPOSITORY[:TAG]]

List images

Options:
  -a, --all             Show all images (default hides intermediate images)
      --digests         Show digests
  -f, --filter filter   Filter output based on conditions provided
      --format string   Pretty-print images using a Go template
      --no-trunc        Don't truncate output
  -q, --quiet           Only show numeric IDs
```

Figura 13: Comandos sobre imágenes de Docker

De esta manera se muestra tanto el uso del comando como los distintos argumentos que se pueden introducir para el comando `images` junto con una breve descripción.

A continuación, se muestran y explican de manera simple (en comparación con la documentación oficial) los comandos más utilizados. Ya que *Docker* está en constante crecimiento se recomienda tanto para esta sección como para todas las posteriores consultar la documentación oficial por si ocurre algún cambio.

Se han dividido los comandos en distintas secciones. Por un lado, se encuentra el comando principal y la base de *Docker,* utilizado para iniciar un contenedor. el comando `run`. Después se tienen comandos útiles para la gestión de los contenedores ya que permiten la interactuación entre el cliente y el contenedor, comandos de información tanto de *Docker* como de los contenedores, que muestran opciones, configuración y el estado actual y por último, comandos acerca de las imágenes y el registro donde se almacenan.

```
$ docker run
```

El comando **RUN** es el más importante ya que inicia un contenedor y el único requisito es que se especifique una imagen, el resto de opciones no son obligatorias. Cualquier opción, comando, argumento, etc. que se indique sobrescribirá a la configuración anterior o a la implementada en el *Dockerfile*. Estas opciones permiten configurar la ejecución, la configuración de la red y asignar privilegios y recursos al contenedor.

Más comandos básicos para distintas utilidades son[16]:

GESTIÓN DE CONTENEDORES	
`$ docker attach`	Permite al usuario ver o interactuar con el proceso principal, entradas, salidas y errores, dentro del contendor.
`$ docker create`	Crea un contenedor desde una imagen pero no lo inicializa, para ello es necesario el comando **docker start**.
`$ docker cp`	Copia archivos y directorios entre el contenedor y el equipo donde se ejecuta.
`$ docker exec`	Ejecuta un comando en el interior del contenedor especificado
`$ docker kill`	Envía una señal al proceso principal (PID 1) del contenedor y devuelve el ID. Por defecto la señal enviada es *SIGKILL*, lo que provoca que se elimine, pero se puede especificar otra señal haciendo uso de los argumentos disponibles.
`$ docker pause`	Suspende todos los procesos dentro del contenedor especificado. Para volver a la ejecución es necesario el uso del comando **docker unpause**. *Docker* realiza la pausa de los procesos internamente con la funcionalidad de Linux *cgroups freezer*.
`$ docker restart`	Reinicia uno o más contenedores indicados. A efectos prácticos es igual que ejecutar el comando **docker stop** seguido por **docker start**.
`$ docker rm`	Elimina uno o más contenedores indicados devolviendo el nombre o ID si se realiza correctamente. Dos de los argumentos que permite son **-f** para forzar la detención del contenedor, en caso que esté en ejecución y **-v** para eliminar los volúmenes asociados.
`$ docker start`	Inicia un contenedor parado. Un contenedor puede estar parado si se ha indicado su detención mediante **docker stop** o si se ha creado con **docker create** pero no ha empezado su ejecución.
`$ docker stop`	Detiene un contenedor sin eliminarlo. Con este comando el contenedor en concreto recibe el estado "*SIGTERM*" y después de un tiempo (periodo de gracia) recibe la señal "SIGKILL" lo que termina con el contenedor. Con el argumento **-t** se puede definir el tiempo de espera necesario para eliminar el contenedor.

[16] Use the Docker command line. (2019). Recuperado de https://docs.docker.com/engine/reference/commandline/cli/

INFORMACIÓN DE DOCKER	
`$ docker info`	Muestra por pantalla información acerca de *Docker* y el equipo donde está instalado.
`$ docker help`	Muestra información sobre el uso de las posibles funciones que tiene *Docker.* La salida por pantalla es la misma que si se introduce el comando **docker --help**
`$ docker version`	Muestra por pantalla la versión tanto del cliente como del servidor de *Docker.*
INFORMACIÓN DE CONTENEDORES	
`$ docker logs`	Muestra los *logs* de un contenedor. Se muestran las salidas **STDOUT** y **STDERR** que se hayan producido dentro del contenedor.
`$ docker port`	Enumera los puertos mapeados en el contenedor.
`$ docker ps`	Muestra información de alto nivel sobre los contenedores que se encuentren en el sistema como nombre, ID y estado.
SOBRE IMÁGENES	
`$ docker build`	Construye una imagen a partir de un *Dockerfile*.
`$ docker commit`	Crea una nueva imagen a partir de los cambios de un contenedor específico.
`$ docker export`	Exporta el contenido de los archivos del sistema del contenedor como un archivo **tar**.
`$ docker history`	Muestra la historia de una imagen.
`$ docker images`	Lista todas las imágenes disponibles en caché en el sistema mostrando información como nombre del repositorio, etiqueta y tamaño.
`$ docker import`	Importa el contenido de un archivo comprimido para crear una imagen.
`$ docker rmi`	Elimina una o más imágenes especificadas por ID o nombre de repositorio y etiqueta.
`$ docker tag`	Asocia un repositorio y una etiqueta con una imagen identificando así cada imagen. Si no se proporciona etiqueta se entiende que es *latest*.
SOBRE REGISTROS	
`$ docker login`	Inicia sesión en el registro de *Docker*. Por defecto el registro es **Docker Hub** pudiendo asignarse otro.
`$ docker logout`	Cierra la sesión en el registro de *Docker*. Por defecto el registro es **Docker Hub** pudiendo asignarse otro.
`$ docker pull`	Descarga la imagen especificada del registro, que está determinado según el nombre de la imagen. Si no se indica etiqueta se descargará la imagen *latest*.
`$ docker push`	Sube una imagen al registro. Si no se especifica etiqueta se subirán todas las imágenes, no solo la marcada como *latest*.

Tabla 2: Comandos básicos de Docker

3.3. CONECTAR CONTENEDORES

Uno de los motivos del auge de *Docker* es la posibilidad de conectar contenedores, no solo entre ellos sino también con recursos externos que no tienen que ser obligatoriamente de *Docker* ya que los contenedores y servicios no necesitan saber si están implementados en un entorno de *Docker* o si lo están los servicios con los que se comunican. Lo que facilita que, por ejemplo, distintos contenedores se puedan comunicar entre ellos aunque ejecuten sistemas operativos distintos[17].

Docker proporciona tres tipos de red para administrar las comunicaciones, tanto dentro de los contenedores como entre ellos: *bridge*, *host* y *none*.

La red *bridge* se utiliza para comunicar aplicaciones independientes entre contenedores, *host* para eliminar el aislamiento de red del contenedor y utilizar la red del propio equipo donde se ejecuta y *none* para desactivar todas las funciones de red en el contenedor.

Por defecto, al crearse los contenedores se conectan a la red *bridge* lo que hace que a cada uno de ellos se le asigne una interfaz virtual y dirección IP privada por lo que el tráfico que pasa a través de la interfaz principal se conecta a la interfaz *docker0* del *host*. También es posible conectar varios contenedores entre sí dentro de una misma red *bridge* a través de las direcciones IP correspondientes.

La comunicación entre el equipo donde se ejecutan los contenedores y los tres modos de red se muestra en la Figura 14. Tanto con la red *host* como con *bridge* los contenedores hacen uso de las interfaces correspondientes para comunicarse entre ellos y con el exterior, mientras que la red *none* no interactúa con ninguna de las interfaces del equipo local[18].

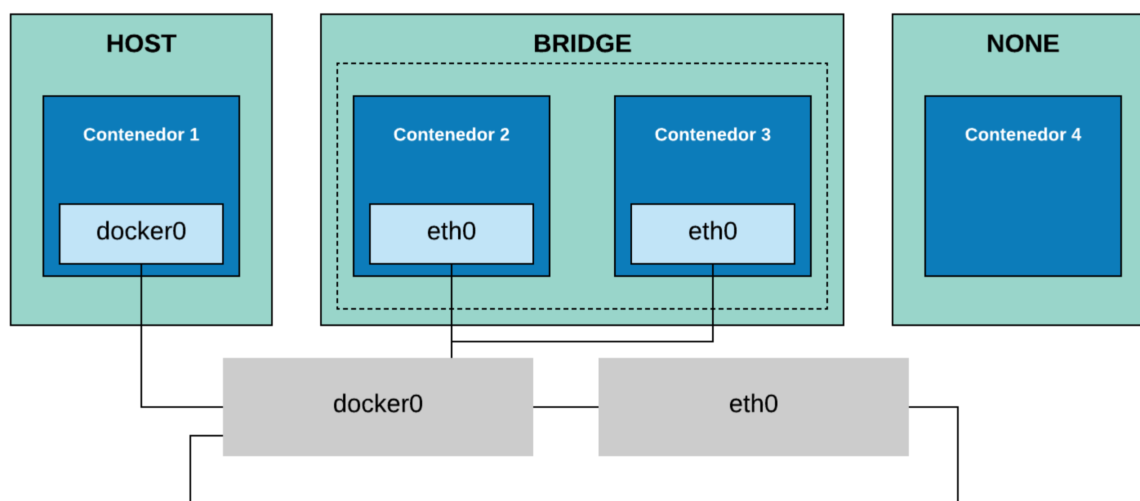

Figura 14: Comunicación entre contenedores

También es posible abrir un puerto específico dentro del contenedor hacia el exterior haciendo uso del comando `--expose` e indicando el puerto, al igual que es posible mapear un puerto del equipo donde se ejecuta *Docker* con un puerto de cualquier contenedor haciendo uso del comando `--publish [host] : [container]` de tal forma que es posible acceder desde el navegador local al contenedor. Por ejemplo, mapeando el puerto 80 del equipo host con el 80 del contenedor, al acceder al puerto local se estaría accediendo al contenedor pudiendo así ver el contenido desde el navegador.

[17] Networking Overview. (2019). Recuperado de https://docs.docker.com/network/
[18] Saito, H., Lee, H., & Wu, C. (2017). *DevOps with Kubernetes*. Birmingham, Reino Unido: Packt Publishing Ltd.

Conociendo el funcionamiento de los contenedores entre ellos y con el entorno, así como los comandos básicos, el siguiente paso sería levantar y ejecutar los contenedores deseados. Como se explica anteriormente, para la ejecución de un contenedor es necesaria una imagen y la definición de dicha imagen se hace mediante un archivo llamado *Dockerfile.*

3.4. DOCKERFILE

Un *Dockerfile* es un archivo de texto plano que contiene una serie de comandos definidos por el usuario para crear la imagen que utilizará el contenedor en su ejecución[19].

Para la construcción de la imagen a partir de un *Dockerfile* y un contexto (conjunto de archivos en una ubicación especificada mediante las etiquetas **PATH** -archivos locales- o **URL** -repositorio GIT-) se hace uso del comando *docker build.* El contexto se procesa de forma recursiva por lo que únicamente es necesario referenciar la carpeta superior para tener también las subcarpetas (o subdirectorios en GIT) incluidos. Si en lugar de utilizar un directorio se quiere seleccionar un archivo en concreto se debe utilizar instrucción **COPY**. Cabe destacar que todo el proceso de construcción es ejecutado por el motor de *Docker* y no por el CLI. En la Figura 15 se muestran los pasos que se ejecutan.

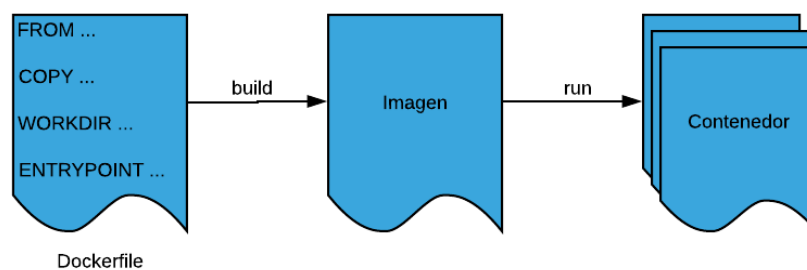

Figura 15: Construcción de un contenedor

Para un mejor rendimiento *Docker* permite utilizar un archivo *.dockerignore* en el que todos los elementos definidos en él no serán tratados en la construcción de la imagen, así es posible seleccionar directorios enteros aunque haya un archivo específico que no se quiera agregar.

El comando de construcción se puede ejecutar desde el mismo directorio donde se encuentre el *Dockerfile* o desde otro siempre y cuando se referencie el archivo para la construcción haciendo uso del comando *-f* (*--file*) seguido de la ruta al archivo. También es posible asignar etiquetas a las imágenes para tenerlas localizadas con el comando *-t* (*--tag*) quedando la sintaxis final de la forma:

```
$ docker image build --file <ruta_al_Dockerfile> --tag
<REPOSITORIO>:<ETIQUETA>
```

Esto significa que *Docker* construirá la imagen creada en el archivo *Dockerfile* en la ruta indicada y le añadirá un nombre específico dado por el repositorio y la etiqueta.

Durante la construcción de la imagen el motor de *Docker* ejecuta las instrucciones una a una de forma independiente comprobando que el resultado es correcto y no ha habido ningún problema. Para cada instrucción se crea una capa que permite reutilizarlas si se encuentran en caché, acelerando significativamente el proceso de construcción. Por ejemplo, una instrucción que requiera de una imagen del registro en la nube supondría una gran carga de trabajo si en cada ejecución es necesario descargarla. Para ello se utilizan los datos en caché, así si la imagen ha sido descargada en una ejecución anterior se utilizará directamente. Cada vez que se hace uso de los datos almacenados en caché se muestra un texto en la consola para que el usuario sea consciente de ello.

[19] Dockerfile Reference (2019). Recuperado de https://docs.docker.com/engine/reference/builder/

3.4.1. SINTAXIS DE UN DOCKERFILE

Al igual que un lenguaje de programación cualquiera, el *Dockerfile* necesita una sintaxis específica para poder ser traducido por el motor de *Docker* y así crear la imagen de forma correcta según las acciones deseadas por el usuario.

Existen múltiples instrucciones aceptadas por el motor de *Docker*, a continuación, se muestran las más básicas y útiles para el usuario.

- La instrucción `FROM` inicia una nueva construcción del contenedor estableciendo la imagen sobre la que se ejecutará el contenedor para las instrucciones siguientes. Un *Dockerfile* válido debe empezar por esta instrucción y puede referirse a cualquier imagen válida, tanto de repositorios públicos como privados. Para una correcta escritura debe especificarse el nombre de la imagen, mientras que la etiqueta para indicar la versión elegida es opcional, teniendo la estructura **IMAGEN:ETIQUETA** (por ejemplo *Ubuntu:14.04* para utilizar la imagen de Ubuntu, versión 14.04). Si se ignora la etiqueta se utilizará ***latest*** (última versión) por defecto. También es posible asociarle un nombre a la imagen con la palabra **AS** siendo la sintaxis final:
    - **FROM** <imagen>[:<tag>] [**AS** <nombre>]
- La instrucción `RUN` ejecuta dentro del contenedor el comando indicado. Existen dos formas correctas de escribir el comando:
    - ***RUN*** *<comando>*
      De esta manera el comando se ejecuta en un intérprete de comandos, */bin/sh -c* en *Linux* y *cmd/S/C* en *Windows*.
    - ***RUN*** *["ejecutable", "parámetro1", " parámetro2"]*
      Se hace uso de *exec* directamente evitando así llamar al intérprete de comandos. Para ello se genera un array en formato *JSON* que se ejecuta en el interior del contenedor, por lo que es necesario el uso de comillas dobles en vez de simples.
- `CMD` establece valores predeterminados para la imagen construida sin ejecutarlos durante la construcción. Solo puede haber una instrucción en el archivo, siendo la última la que prevalecerá en caso de haber más de una. Las diferentes formas de utilizar el comando son casi iguales que para el comando **RUN**:
    - ***CMD*** *["ejecutable"," parámetro1"," parámetro2"]*
      Forma principal de uso, de esta manera se llama directamente a la ejecución *exec*.
    - ***CMD*** *["parámetro1"," parámetro2"]*
      Uso como parámetro por defecto de **ENTRYPOINT**.
    - ***CMD*** *comando parámetro1 parámetro2*
      De esta manera se hace uso del intérprete de comandos.
- `ENTRYPOINT` determina el ejecutable predeterminado para una imagen, es decir, cuando una contenedor se inicializa, ejecuta el archivo indicado. Al igual que en los anteriores comandos la sintaxis diferencia entre el uso o no del intérprete de comandos.
    - ***ENTRYPOINT*** *["ejecutable", " parámetro1", " parámetro2"]*
      Llamada a *exec* directamente.
    - ***ENTRYPOINT*** *comando parámetro1 parámetro2*
      Haciendo uso de un intérprete de comandos.

  El comando **ENTRYPONIT** puede ser combinado con **CMD** y **RUN**, teniendo un comportamiento diferente dependiendo de como se escriba. Hay tres posibles escenarios:

    1. **ENTRYPOINT** hace uso de un intérprete de comandos (*shell*): El comando **CMD** y los parámetros de ***docker run*** se ignoran.

2. **ENTRYPOINT** hace uso de *exec* y ***docker run*** tiene argumentos*:* Los parámetros del **CMD** se sobrescriben por los de ***docker run***.
3. **ENTRYPOINT** hace uso de *exec* y solo tiene valores configurados el comando **CMD**: Se combinan ambos comandos.

Por lo general se utiliza **ENTRYPOINT** para apuntar a la aplicación principal que se desea ejecutar y **CMD** para los parámetros predeterminados.

- **ENV** establece las variables de entorno de una clave asociándole un valor determinado tanto para las siguientes instrucciones durante la construcción como para la imagen final construida. Se puede escribir de dos maneras:
    - ***ENV** <key> <value>*
      Se establece un único valor definido después del espacio para una única *key*.
    - ***ENV** key1=value1 key2=value2*
      Se establecen múltiples valores para múltiples claves.
- La instrucción **LABEL** funciona de manera similar a **ENV** pero almacenando el par clave-valor únicamente en la sección de metadatos a la imagen construida pudiendo ser utilizados estos datos únicamente por otro programas que no sean el que se ejecuta dentro del contenedor. *Docker* proporciona el comando **--filter** para filtrar objetos según su etiqueta. La sintaxis de uso viene definida por:
    - ***LABEL** <key>=<value> <key>=<value> <key>=<value> ...*
- **EXPOSE** actúa de forma idéntica al comando propio de *Docker* **--expose**, informando al motor de *Docker* que el contenedor escucha en los puertos indicados. Por defecto se utiliza como protocolo **TCP** pero se puede especificar **UDP** siendo la sintaxis:
    - ***EXPOSE** <port> [<port>/<protocol>...]*
- La instrucción **USER** cambia al usuario (o UID) por el indicado para ejecutar las siguientes acciones de **RUN**, **CMD** y **ENTRYPOINT**. De forma opcional también se puede establecer, además del usuario, el grupo de usuarios (o GID). Si el usuario no existe en la imagen será necesario añadirlo previamente con la instrucción **ADDUSER**. La correcta escritura del comando es, a elegir según las necesidades:
    - ***USER** <usuario>[:<grupo>]*
    - ***USER** <UID>[:<GID>]*
- **WORKDIR** establece el directorio de trabajo para las siguientes instrucciones **RUN**, **CMD**, **ENTRYPOINT**, **ADD** y **COPY** definidas en el *Dockerfile*. Si el directorio de trabajo indicado no existe se creará aunque no sea utilizado por ninguna instrucción. A efectos prácticas funciona igual que el comando *cd* en sistemas UNIX:
    - ***WORKDIR** /ruta/al/directorio*
- **COPY** copia los archivos o directorios desde el origen al destino, dentro del contenedor, indicados. Se pueden especificar múltiples recursos de origen para copiar entendiéndose todas las rutas como relativas al origen del contexto de la aplicación:
    - ***COPY** <origen> ... <destino>*
- **ADD** tiene una función similar a la instrucción **COPY**, mover archivos indicados a la imagen del contenedor. La diferencia es que **ADD** no solo copia archivos que se encuentren en el host, sino que también puede ser una URL o un archivo comprimido, descargándolo en el primer caso y descomprimiendo en el segundo, siempre en la ruta destino indicada.
    - ***ADD** <origen> ... <destino>*

Estas instrucciones no son las únicas pero sí las más usadas por los usuarios y las que abarcan las opciones más básicas. Existen múltiples instrucciones más que permiten realizar todos los cambios y configuraciones necesarias dentro del contenedor durante y después de su construcción.

3.4.1.1. BUENOS HÁBITOS

Como con cualquier otro elemento dentro del desarrollo de software, mantener unos buenos hábitos a la hora de escribir y organizar los archivos *Dockerfile* es importante para tener una estructura clara y definida que ayude a conseguir el objetivo final de una manera más simple. Las acciones que más ayudarán a mantener el proyecto estructurado son:

- Utilizar el archivo *.dockerignore* antes mencionado de forma habitual siempre que sea necesario. De igual manera que al utilizar GIT el archivo *.gitignore* se hace imprescindible, también debe serlo a la hora de crear contenedores.
- Tener como máximo un único *Dockerfile* por carpeta para organizar mejor los contenedores.
- Utilizar control de versiones en los *Dockerfile* al igual que se utiliza con cualquier otro lenguaje de programación. Al fin y al cabo un *Dockerfile* es un archivo que crea un comportamiento determinado en un contenedor, de igual manera que un archivo, por ejemplo, *.java* genera un comportamiento. Si es normal y lógico utilizar control de versiones para un archivo *.java* también lo es para un *Dockerfile.*
- Utilizar los mínimos recursos posibles en la creación de las imágenes. Uno de los mayores retos es optimizar las imágenes creados al máximo utilizando los recursos y paquetes estrictamente necesarios.
- Ejecutar una única aplicación por contenedor. Cada vez que se necesite una nueva aplicación es mejor y más práctico iniciar un nuevo contenedor y ejecutarla en su interior en lugar de introducir cambios en una imagen ya creada para introducir el nuevo programa.

3.5. ORQUESTACIÓN MULTICONTENEDOR

Una vez vista la manera de crear un único contenedor con la configuración predeterminada, el siguiente paso viene dado por la necesidad de realizar este proceso de manera repetitiva para múltiples aplicaciones de manera simultánea. Es por ello que aparece el término “orquestación de contenedores” u “orquestación multicontenedor”.

La orquestación de contenedores es la automatización de todos los aspectos de la coordinación y gestión de contenedores, además de la organización, que se centra en gestionar el ciclo de vida de los propios contenedores y sus entornos dinámicos[20].

Según lo expuesto hasta ahora podemos saber cómo iniciar los contenedores necesarios cada uno por separado, con los requisitos y las funcionalidades correspondientes necesarias, además de la conexión de red para la comunicación entre ellos etc. Pero iniciar de esta manera los contenedores no es la mejor solución; un contenedor puede necesitar más tiempo que otros para iniciar y causar errores, por ejemplo. Una solución puede ser lanzar los contenedores de forma manual asegurándose que todos los que tienen relación se inician correctamente en el orden adecuado. Para evitar que el programador haga este trabajo, ya que puede cometer errores además de ser una tarea tediosa, *Docker* permite configurar de forma automática el despliegue de varios contenedores con una funcionalidad llamada *docker compose*.

3.5.1. DOCKER COMPOSE

Compose es una herramienta para definir y ejecutar aplicaciones *Docker* multicontenedor. Está integrada en la distribución de *Docker* y ejecuta los contenedores definidos en un archivo llamado

[20] What is Container Orchestration?. (2019). Recuperado de https://avinetworks.com/glossary/container-orchestration/

docker-compose.yaml (o *.yml*) de forma similar a un *Dockerfile* de manera que con un único comando se pueden crear e iniciar todos los servicios deseados[21].

Esta herramienta puede ser utilizada en todos los entornos y etapas del desarrollo de software, producción, desarrollo, testing, etc. y permite no centrarse únicamente en contenedores de forma individual sino en todo el entorno y las interacciones entre los servicios manteniendo el aislamiento entre los contenedores y la posibilidad de escalarlos de manera individual. Su uso se define en tres pasos:

1. Definir el entorno (imagen) del servicio mediante un archivo *Dockerfile*, si es necesario.
2. Definir los servicios en el archivo *docker-compose.yml* para ejecutarse cada uno de ellos en un entorno aislado
3. Ejecutar `docker-compose up` para que la herramienta *Compose* inicie el trabajo y ejecute todas las aplicaciones.

Para su instalación primero es necesario descargar el paquete oficial desde el repositorio de GitHub de *Docker*:

```
root@ubuntu:/# sudo curl -L
"https://github.com/docker/compose/releases/download/1.24.0/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Y después hacer el archivo ejecutable:

```
root@ubuntu:/# sudo chmod +x /usr/local/bin/docker-compose
```

3.5.1.1. SINTAXIS

Un archivo de *docker-compose* tiene un aspecto similar al mostrado a continuación:

```
version: '3'

services:

 web:

    build: .

    volumes:

    - .:/code

    - logvolumen01:/var/log

    links:

    - redis

  redis:

    image: redis

  volumes:

    logvolumen01: {}
```

[21] Docker Compose. (2019). Recuperado de https://docs.docker.com/compose/

Todos los archivos deben tener una estructura similar para poder ser leídos correctamente por el motor de *Docker*. Cabe destacar el proceso de construcción donde se indica `build: .` lo que significa que se utiliza el archivo *Dockerfile* ubicado en el mismo directorio. Si se indicase la ruta completa podría hacerse referencia a un archivo en cualquier otro directorio del ordenador. El archivo necesario para la ejecución de *Compose* tiene tres partes diferenciables.

La primera parte consta de la versión a utilizar. Esta versión será sobre la que trabaje la API de *Docker* para realizar las peticiones.

```
version: "3"
```

La segunda parte son los servicios definidos siguiendo la siguiente estructura.

```
services:
--> nombre del contenedor
----> opciones del contenedor
--> nombre del contenedor
----> opciones del contenedor
```

La sintaxis utilizada para definir los servicios es muy similar a la utilizada en el comando `docker run` para iniciar contenedores compartiendo *flags* y palabras reservadas, aunque con diferencias notables. Por ejemplo, mediante el comando `docker run` no se puede especificar la imagen pero sí volúmenes o puertos a exponer. La mayor diferencia reside en el uso de *depends_on* que no existe en los comandos de *Docker* ya que organiza las ejecuciones de los contenedores de manera que no se inicialice uno hasta que no lo hagan el resto de contenedores de los que depende. En el archivo *docker-compose* cada opción debe aparecer debajo del contenedor sobre el que se quiere ejecutar.

La tercera y última parte incluye los demás factores que pueden estar en el entorno de los contenedores, ya sean volúmenes (como en el ejemplo) o redes creadas, entre otros.

```
volumes:
  logvolume01: {}
```

3.5.1.2. COMANDOS

Los comandos mostrados a continuación son los más utilizados y, aunque muchos de ellos sean similares a los comandos de *Docker* o tengan un nombre muy intuitivo, conviene tener claro el modo de uso para todos ellos.

- `up`: Inicia todos los contenedores definidos en el archivo *docker-compose.yml* y muestra la salida por pantalla. Para ejecutar el proceso en un segundo plano se utiliza el argumento *-d*.
- `build`: Reconstruye una imagen creada por un *Dockerfile.* El comando ***up*** no construye las imágenes a menos que no existan por lo que cada vez que se actualice la imagen es necesario construirla nuevamente.
- `ps`: Proporciona información sobre el estado de los contenedores que se hayan construido con *Compose.*
- `run`: Ejecuta el comando especificado una única vez sobre un servicio en concreto y sus enlaces.
- `logs`: Muestra por pantalla *logs* de los contenedores construidos por *Compose*.
- `stop`: Detiene los contenedores sin eliminarlos.

- `rm`: Elimina los contenedores detenidos. Para eliminar los que se encuentran en ejecución será necesario el uso de *-f* y así forzar la acción.

Una ejecución normal comienza por introducir el comando `docker-compose up`, que iniciará el proceso de creación de los contenedores. Para buscar errores y verificar el funcionamiento de la aplicación se pueden utilizar los comandos `docker-compose logs` y `docker-compose ps.`

Si se realizan cambios en el código es necesario reconstruir los contenedores mediante `docker-compose build` seguido por `docker-compose up` para reemplazar a los anteriores manteniendo los volúmenes y datos persistentes. Si en lugar del código de la aplicación únicamente se ha cambiado la configuración de los contenedores, es decir, si en lugar de cambiar la imagen se ha cambiado el documento *docker-compose.yml* únicamente es necesario ejecutar el comando `docker-compose run` [22].

Una vez terminada la ejecución deseada se puede utilizar el comando `docker-compose stop` para detener los procesos. Estos contenedores detenidos se pueden iniciar nuevamente con `docker-compose start` o bien eliminarlos de manera permanente con el comando `docker-compose rm`.

[22] Mouat, A. (2015). *Using Docker*. Sebastopol, CA: O'Reilly Media Inc.

CAPÍTULO 4

ORQUESTACIÓN DE CONTENEDORES: KUBERNETES

4. ORQUESTACIÓN DE CONTENEDORES: KUBERNETES

En el capítulo anterior se muestra una manera de orquestar varios contenedores simultáneamente. Esta orquestación es útil a la hora de la creación, pero *Docker* no tiene las suficientes herramientas para garantizar la facilidad de uso en otros aspectos dentro del desarrollo del software. Para ello se hace uso de *Kubernetes*, que crea un clúster de contenedores con diferentes objetos propios para facilitar el proceso al desarrollador permitiendo desplegar, controlar y escalar las distintas aplicaciones de manera automática.

Kubernetes tiene diferentes niveles de abstracción dentro del clúster mediante los cuales se representa el estado del sistema: aplicaciones desplegadas en contenedores, cargas de trabajo, recursos de almacenamiento o de red necesarios e información adicional del estado del clúster en todo momento. Estas abstracciones están representadas por objetos y la manera que tiene el usuario de interactuar con ellos es mediante la API de *Kubernetes*. Los objetos básicos y más importantes son *pod*, servicio, volumen y espacio de nombres (*namespace*). Además, existe un nivel superior de abstracción en *Kubernetes* donde se encuentran los llamados **controladores** que se basan en los objetos básicos y proporcionan funcionalidades adicionales sobre ellos entre los que destacan *ReplicaSet* y *Deployment*[23].

4.1. OBJETOS DE KUBERNETES

Los objetos de *Kubernetes* son entidades persistentes en el sistema del clúster. *Kubernetes* utiliza estas entidades para mostrar el estado del clúster, específicamente:

- Qué aplicaciones se están ejecutando en contenedores y el nodo en el que lo hacen
- Los recursos disponibles para esas aplicaciones
- Las políticas y reglas asociadas a esas aplicaciones.

Cada vez que un objeto se crea en el clúster *Kubernetes* garantiza que siga existiendo y mantenga su funcionalidad.

Un objeto dentro del clúster queda definido por dos factores, la especificación y el estado. La especificación describe el estado deseado, es decir, las características y configuración que se quieren tener en el objeto mientras que el estado describe el punto en el que se encuentra el objeto en el momento actual. Estos dos factores están suministrados y actualizados por *Kubernetes* que se encarga de, en todo momento, coincidan el estado actual igual y el estado especificado o deseado.

Esta especificación del estado del objeto deseada es necesario realizarla al motor mediante la línea de comandos o mediante un archivo de configuración. Por la gran cantidad de información que suele ir asociada a cada despliegue no es cómodo ni práctico realizarlo directamente mediante comandos excepto en situaciones muy concretas, por lo que la mejor solución es configurar un archivo ***.yaml***.

Un ejemplo de archivo de configuración es el mostrado a continuación:

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment
```

[23] Kubernetes Components. (2019). Recuperado de https://kubernetes.io/docs/concepts/overview/components/

```
    labels:
      app: nginx
  spec:
    replicas: 3
    selector:
      matchLabels:
        app: nginx
    template:
      metadata:
        labels:
          app: nginx
      spec:
        containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
          - containerPort: 80
```

Crear un objeto mediante un archivo similar al anterior permite crear multitud de variantes distintas y realizar casi cualquier función que queramos gracias a la versatilidad de los distintos objetos ya que no todos ellos necesitan toda la información mostrada; los únicos campos obligatorios comunes a todos son:

- `apiVersion`: Especifica qué versión de la API de *Kubernetes* se va a utilizar para crear el objeto.
- `kind`**:** Especifica qué tipo de objeto se quiere crear.
- `metadata`**:** Es un dato único que permite diferenciar al objeto incluyendo una cadena de caracteres como nombre (**name**) o id de usuario (**UID**) y de manera opcional también un **namespace**.

También debe rellenarse el campo de especificación del objeto (**spec**) con la estructura acorde al objeto que se esté creando, teniendo cada objeto sus propias especificaciones.

Para conocer los campos necesarios y especificarlos en el archivo se puede consultar la documentación oficial de *Kubernetes* referente a la API (https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.14/). De igual manera se explica a continuación el funcionamiento y la lógica de los objetos básicos dentro del clúster.

4.1.1. POD

Un *pod* es el componente más básico de *Kubernetes*, el objeto más pequeño y simple que se encuentra en el clúster y representan cada uno a un proceso en ejecución dentro del clúster.

Cada uno de los *pods* existentes encapsulan la aplicación deseada, recursos de almacenamiento, dirección IP y configuración adicional sobre cómo debe ejecutarse el contenedor. Un *pod* representa una única instancia de una aplicación en *Kubernetes* , pudiendo ser de uno o más contenedores que comparten recursos.

Cuando un *pod* muere por alguna razón no es posible volver a recuperarlo por él mismo, sino que es el controlador de *Kubernetes* quien decide si crea o no uno nuevo para cumplir con el número total de *pods* deseados por el usuario.

Los recursos a los que deben acceder el contenedor o los contenedores existentes son de red y de almacenamiento, en cuanto a la red cada *pod* tiene asignada una dirección IP única y en su interior cada contenedor utiliza la misma red, tanto IP como puerto. Si existe más de un contenedor la conexión entre ellos se realiza a través de **localhost** mientras que si la conexión es con una entidad exterior al *pod* son los contenedores los que se coordinan para utilizar los recursos de red compartidos. Para el almacenamiento cada *pod* puede especificar un almacenamiento (llamado volumen) compartido al que pueden acceder todos los contenedores existentes y así compartir los datos necesarios. Los volúmenes creados pueden ser persistentes para guardar la información necesaria, aunque el *pod* tenga que reiniciarse.

4.1.2. VOLUMEN

Los archivos en disco relacionado con los contenedores que se encuentran en ejecución en el interior de los distintos *pods* son efímeros, lo que presenta dos problemas principales. Primero, cuando un contenedor detiene su ejecución es reiniciado, pero pierde todo el contenido que pudiese tener ya que empieza con la configuración inicial y segundo, cuando dos contenedores están en ejecución simultáneamente en un mismo *pod*, normalmente es necesario que cambien información entre ellos. Estos dos problemas básicos para el ciclo de vida en aplicaciones no triviales se resuelven en *Kubernetes* gracias al uso de volúmenes.

Un volumen en *Kubernetes* puede considerarse como un directorio al que pueden acceder los distintos contenedores dentro de un *pod* y así guardar la información necesaria. Un volumen, generalmente, tiene el mismo periodo de vida que el *pod* que lo contiene, así aunque los contenedores se reinicien la información sigue presente pero se perderá si el *pod* deja de existir. *Kubernetes* permite crear distintos tipos de volúmenes; cómo se originan, quien los maneja y su contenido depende exclusivamente del tipo de volumen utilizado. Los más utilizados son:

- **emptyDir**: Tipo básico de volumen creado cuando un *pod* es recién asignado a un nodo. Inicialmente es un directorio vacío (como indica su nombre) y los contenedores escriben la información necesaria. El volumen permanece activo todo el tiempo que lo esté el *pod* que lo contiene y una vez que éste termine también lo harán los datos.
- **nfs**: Es un tipo de volumen que permite que un recurso compartido de NFS (Network File System) existente se monte en el *pod*. Cuando un *pod* termina su ejecución el volumen en vez de eliminarse se desmonta por lo que la información que contenga puede ser utilizada por otros *pods* incluso de manera simultánea.
- **persistentVolumeClaim**: Se utiliza para montar un volumen persistente, que son una forma de utilizar espacio de almacenamiento de manera duradera.
- **secret**: Utilizado para pasar información confidencial, como pueden ser contraseñas, claves de acceso o *tokens*, a los *pods*. Se almacenan en formato de pares clave-valor haciendo uso de *tmpfs*, que utiliza memoria volátil.

4.1.3. SERVICIO

Un servicio en *Kubernetes* es una capa de abstracción utilizada para enrutar el tráfico a los *pods* correspondientes por lo que no es necesario buscar la dirección IP de cada uno de ellos soportando TCP y UDP. Generalmente se utilizan etiquetas para seleccionar los *pods* que se enrutan por lo que lo único que necesita el servicio es que dicha etiqueta coincida independientemente de la manera de la que han sido creados los *pods*[24].

Existen tres tipos principales de servicios: *ClusterIP*, *NodePort* y *LoadBalancer*, esquematizados en la Figura 16.

- **ClusterIP**: Es el tipo de servicio predeterminado; expone el servicio con una IP interna del clúster por lo que es accesible únicamente por los objetos que se encuentren en el interior del mismo.
- **NodePort**: Expone el servicio en cada nodo haciendo uso de la propia IP del nodo y un puerto estático (el ***NodePort***) creándose automáticamente un servicio *ClusterIP* al que se enruta el servicio *NodePort*. Al exponer el nodo se puede acceder al servicio desde fuera del clúster mediante la ruta **<NodeIP>:<NodePort>**
- **LoadBalancer**: Expone el servicio externamente haciendo uso de un balanceador de carga provisto por un agente externo. Se crean automáticamente los servicios *NodePort* y *ClusterIP* para seguir el camino entre el *pod* y el exterior.

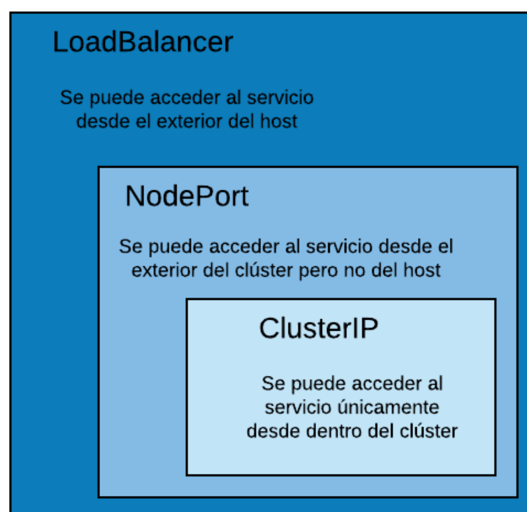

Figura 16: Tipos de servicios en Kubernetes

4.1.4. NAMESPACE

Diseñados para utilizarse en entornos con muchos usuarios distribuidos en varios equipos o proyectos, un *namespace* es considerado un aislamiento entre distintos clústeres virtuales ya que los distintos objetos que existan en un *namespace* son invisibles para los demás. Existen tres *namespaces* por defecto en *Kubernetes* teniendo el usuario la posibilidad de añadir otros propios con los objetos que desee. Estos son:

- **default**: El *namespace* por defecto para los objetos que no se especifica ningún otro.
- **kube-system**: Utilizado por los objetos creados por el sistema de *Kubernetes*.
- **kube-public**: Creado automáticamente y accesible por todos los usuarios. Es utilizado en caso de que un recurso en concreto quiera ser accesible por todo el clúster.

[24] Saito, H., Lee, H., & Wu, C. (2017). *DevOps with Kubernetes*. Birmingham, Reino Unido: Packt Publishing Ltd.

4.1.5. REPLICASET

El objetivo del *ReplicaSet* es mantener un conjunto estable de *pods* en ejecución asegurándose que el número de réplicas especificadas se encuentran en ejecución en cualquier momento.

Un *ReplicaSet* se define con varios campos que le permiten identificar los *pods* a los que debe controlar y una plantilla donde se indican los datos de los nuevos *pods* que debe crear según las réplicas necesarias. Seguidamente crea o elimina los *pods* necesarios para que el número total en ejecución coincida con el número de réplicas deseadas.

4.1.6. DEPLOYMENT

Un *deployment* (despliegue) proporciona actualizaciones para los *ReplicaSet* y para los *pods*, lo que lo convierte en la mejor manera para administrar e implementar aplicaciones en *Kubernetes* ya que se define la actualización deseada y el despliegue interactúa con los elementos necesarios, generalmente *ReplicaSet* y *pods* para realizarla progresivamente de forma automática.

Si una aplicación con múltiples *pods* se ejecuta dentro de un despliegue y uno de los *pods* se elimina, volverá a crearse uno nuevo automáticamente. Esto se debe al trabajo interno que realiza el despliegue, ya que al no coincidir el número de *pods* actual con el deseado, internamente se crea un *ReplicaSet* para asegurar la concordancia.

Como se muestra en la Figura 17 un despliegue está en contacto con los *pods* actuales que mantiene en su ejecución y con los *ReplicaSets* de los que hará uso en el caso que sea necesario escalar la aplicación.

Al ser un proceso automático facilita la implementación de aplicaciones en el clúster.

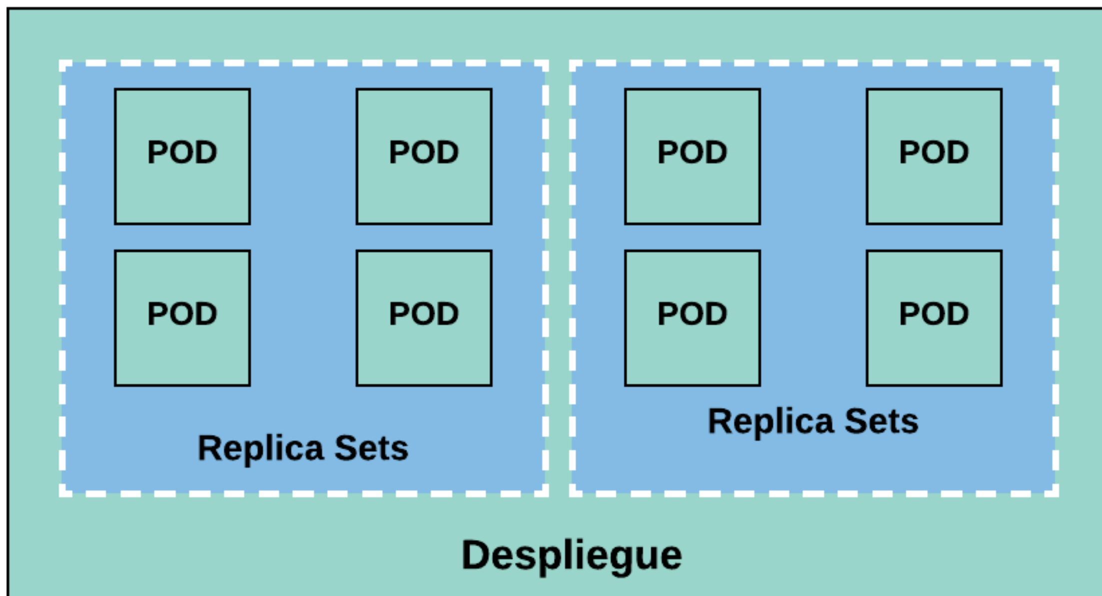

Figura 17: Relación entre Despliegue, ReplicaSet y Pod

4.2. COMPONENTES DE KUBERNETES

En *Kubernetes* se utilizan objetos de la API para conocer el estado del clúster en todo momento, qué aplicaciones se ejecutan en un momento determinado o cuáles se quieren ejecutar, qué imágenes, red o recursos utilizan, cuántas réplicas son necesarias y un largo etcétera.

Una vez especificado el estado deseado del clúster, *Kubernetes* realizará las acciones necesarias para que el estado actual coincida con el deseado, ejecutando tantas tareas como sean necesarias mediante el *Plano de Control*.

El *Plano de Control* es un grupo de procesos que corren en el clúster divididos en dos componentes principales: Maestro y nodos. El maestro es el responsable de mantener el estado deseado del clúster al ejecutar los procesos que controlan y organizan todas las actividades que ocurren en el clúster; de hecho cuando el cliente se comunica con *Kubernetes*, realmente se está comunicando con el nodo maestro. Mientras que el resto de nodos se encargan de ejecutar los contenedores.

El **maestro** incluye distintos procesos indispensables en el corazón de *Kubernetes* como son el servidor API, el controlador principal, el organizador (*scheduler*) y el *etcd*, dispuestos todos ellos tal y como se muestra en la figura 18[25].

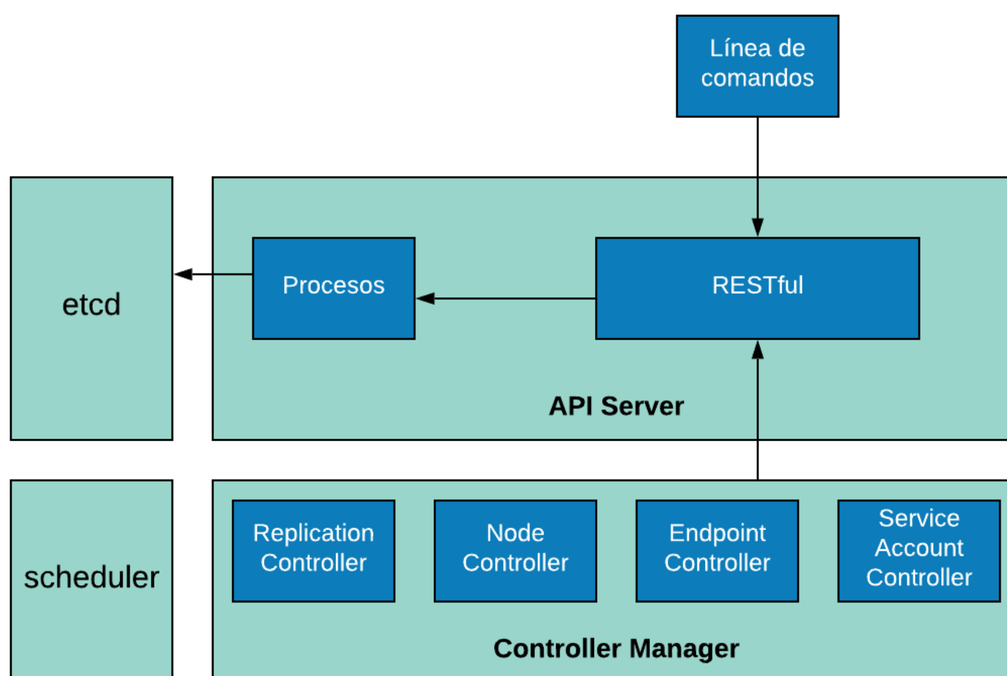

Figura 18: Arquitectura nodo maestro

- El servidor API (*API Server* en inglés) presente dentro del clúster con nombre **kube-apiserver** es el encargado de proporcionar un servidor HTTP o HTTPS con una API RESTful para todos los componentes de la parte del maestro además de leer y actualizar el *etcd*, que es el almacén de datos de *Kubernetes*.

- Controlador principal (*Controller manager*) también llamado **kube-controller-manager** se encarga de ejecutar distintos controladores dentro del clúster:

- Controlador de nodo: Responsable de observar el comportamiento de los nodos y notificar cuando alguno de ellos deja de estar disponible.
- Controlador de replicación: Responsable de mantener el número correcto de *pods*.
- Controlador de *endpoints*: Responsable de gestionar la relación entre servicios y *pods*.
- Cuenta de servicio y controlador de token: Responsable de controlar los *tokens* de acceso a la API y la cuenta predeterminados en el clúster.

- *etcd* se trata de un almacén de pares clave-valor utilizado por *Kubernetes* para guardar todos los datos del clúster.

[25] Kubernetes Components. (2019). Recuperado de https://kubernetes.io/docs/concepts/overview/components/

- El organizador (*scheduler*) o **kube-scheduler** se encarga de asignar un nodo para su ejecución a los *pods* recién creados. Esta asignación se hace en base a distintos factores como el uso de recursos, capacidad o el balanceado de carga necesario, entre otros.

Por otra parte, los **nodos** deben ejecutarse manteniendo los *pods* en ejecución e informando al maestro de su estado. Los procesos que se encuentran en los nodos son *kubelet*, un *proxy* y el software utilizado para la encapsulación en contenedores sobre el que se basa el clúster, que por defecto en *kubernetes* es *Docker*, como se muestra en la Figura 19.

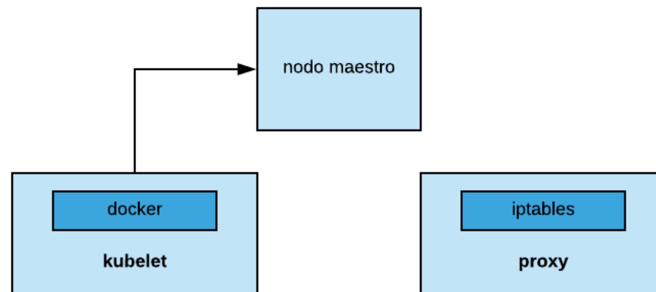

Figura 19: Funcionamiento de los nodos

- *kubelet* es el proceso principal que ejecuta cada uno de los nodos. Se encarga de informar de las actividades de los nodos a **kube-apiserver** periódicamente y si se produce algún error o mantiene su correcta ejecución.

- El *proxy* se encarga del enrutamiento tanto entre los servicios y los *pods* como entre el exterior y los propios servicios.

- *Docker* es el software de encapsulación utilizado por *Kubernetes*.

La interacción entre estos componentes, como se muestra en la Figura 20, es la siguiente; el cliente mediante *kubectl* interactúa con el servidor haciendo uso de la API, que a su vez llama a *etcd* para cargar (GET) o guardar (POST) los valores especificados. El *scheduler* determina qué nodo debe encargarse de la tarea y los controladores monitorizan el estado de las tareas y sus respuestas para responder si fuese necesario. Por otro lado, la API recibe datos de los *pods* mediante *kubelet* que pueden utilizarse, por ejemplo, para la comunicación entre distintos nodos maestros.

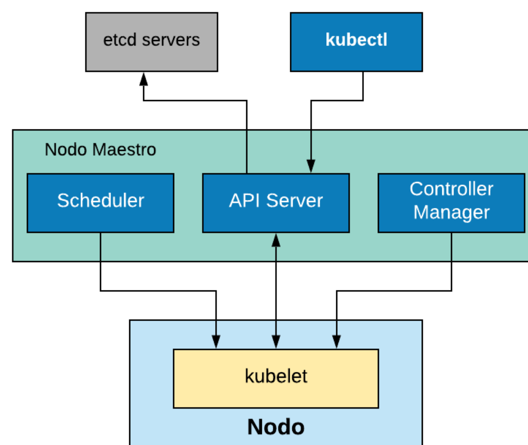

Figura 20: Interacción entre nodos

4.3. EMPEZANDO CON KUBERNETES

Conociendo el funcionamiento básico a nivel teórico de *Kubernetes*, el siguiente paso es crear un entorno que permita utilizar la herramienta. Este entorno se denomina clúster y las opciones para crearlo existen tanto a nivel local como en la nube. Las soluciones a nivel local más utilizadas son *kubeadm*, que permite incluir varios nodos al clúster teniendo así un maestro y diferentes trabajadores, y *Minikube*, explicado más adelante, se compone únicamente de un nodo es la solución ideal para entornos de desarrollo y pruebas fuera del ámbito empresarial.

Sea cual sea la opción elegida un punto en común a todas ellas es la obligación de tener que comunicarse con la API de *Kubernetes* para satisfacer las peticiones. Dicha comunicación se realiza mediante **kubectl**.

4.3.1. INTERACTUAR CON EL CLÚSTER: KUBECTL

La manera que tiene un usuario para interaccionar con el clúster es mediante la API de *Kubernetes*. Para facilitar esta comunicación existe **kubectl**, una herramienta que traduce comandos introducidos por el cliente mediante una interfaz de línea de comandos al motor de *Kubernetes*, siendo su sintaxis de la forma:

```
kubectl [command] [TYPE] [NAME] [flags]
```

Donde:

- `command`: Especifica la operación que se quiere realizar sobre uno o más recursos.
- `TYPE`: Especifica el tipo de recurso (*pod*, servicio, *namespace*…). Puede escribirse en singular, plural o abreviado (po, svc, ns…).
- `NAME`: Especifica el nombre del recurso. Si se omite se mostrarán todos los recursos del tipo indicado
- `flags`: Especifica parámetros de entrada opcionales. Se pueden ver todos los parámetros permitidos con el comando `kubectl options`.

Para poder utilizar esta interfaz es necesario tenerla instalada en el equipo donde se trabaje. *kubectl* se trata de un archivo binario ejecutable por lo que para su instalación lo único necesario será descargar el paquete, hacerlo ejecutable y moverlo al *PATH*, la carpeta ***bin*,** de esa manera será accesible mediante la línea de comandos. Se realiza mediante la serie de comandos:

1. Descargar el paquete:
   ```
   curl -LO https://storage.googleapis.com/kubernetes-
   release/release/$(curl -s
   https://storage.googleapis.com/kubernetes-
   release/release/stable.txt)/bin/linux/amd64/kubectl
   ```
2. Hacer el archivo ejecutable
   ```
   chmod +x ./kubectl
   ```
3. Mover el archivo al *PATH*
   ```
   sudo mv ./kubectl /usr/local/bin/kubectl
   ```
4. Comprobar el correcto funcionamiento
   ```
   kubectl versión
   ```

Al escribir este último comando la salida debe ser similar a la mostrada en la Figura 21

Como se puede observar la parte del cliente funciona correctamente pero el servidor devuelve un error donde indica que no es posible conectarse. Este error es debido a que *kubectl* intenta interaccionar

con un clúster de *Kubernetes* pero no se encuentra ninguno levantando. La solución para esto es crear un clúster y para ello una de las soluciones más simples y eficientes es hacer uso de *Minikube*.

```
root@ubuntu:~# kubectl version
Client Version: version.Info{Major:"1", Minor:"13", GitVersion:"v1.13.1", GitCo
mmit:"eec55b9ba98609a46fee712359c7b5b365bdd920", GitTreeState:"clean", BuildDat
e:"2018-12-13T10:39:04Z", GoVersion:"go1.11.2", Compiler:"gc", Platform:"linux/
amd64"}
The connection to the server 192.168.99.100:8443 was refused - did you specify
the right host or port?
```

Figura 21: Versión utilizada por kubectl

4.3.2. PREPARAR EL ENTORNO: MINIKUBE

La manera más simple de empezar a interaccionar con un clúster de *Kubernetes* es a través de **Minikube**. Es un proyecto oficial de *Kubernetes* que permite ejecutar un clúster de un único nodo en un entorno local. Es una herramienta multiplataforma ya que permite su uso tanto para Windows como Linux y macOS[26].

Dado que *Minikube* configura un clúster de un único nodo existen limitaciones que hacen que la herramienta no sea útil si se requieren orquestar aplicaciones que necesiten carga pesada o en un entorno de producción, sin embargo, es muy útil para desarrollo y testeo de productos software.

Al ejecutar la herramienta lanzará una máquina virtual con *Kubernetes* instalado sobre la que trabajará el clúster a menos que se indique lo contrario con el parámetro **vm-driver=none** lo que hará que *Minikube* utilice directamente el host de *Docker* instalado en el equipo. Este uso es peligroso debido a que se ejecuta el programa como administrador directamente en el equipo, lo que lo hace vulnerable a posibles ataques. Sin embargo, es necesario si se quiere crear el clúster en una máquina virtual ya que no se permite otro grado más de virtualización, es decir, la máquina virtual sobre la que se trabaja no permite a *Minikube* crear una máquina donde ejecutar *Kubernetes*.

Al igual que **kubectl**, *Minikube* también se trata de un archivo binario ejecutable, así que, siguiendo la misma filosofía que anteriormente los pasos a seguir son descargar el paquete y moverlo al *PATH*. Para ello los comandos necesarios son:

1. Descargar el paquete
   `curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64`
2. Hacerlo ejecutable
   `chmod +x minikube`
3. Moverlo al *PATH* (y eliminarlo)
   `sudo cp minikube /usr/local/bin && rm minikube`
4. Comprobar que funciona
   `minikube version`

Una vez instalado podemos comprobar que el sistema reconoce la palabra clave **minikube**. Para iniciar el clúster únicamente es necesario el comando `minikube start`. En el proceso de inicio se descarga la imagen necesaria (si no se encuentra en caché) y se realizan las configuraciones indicadas y en caso de que no se indique ninguna las que *Minikube* realice por defecto.

Por ejemplo, en la Figura 22 se inicia la herramienta sin ninguna configuración adicional y tanto la máquina virtual elegida como los recursos hardware necesarios o la dirección IP del clúster se

[26] Kubernetes, Minikube. (2019). Recuperado de https://github.com/kubernetes/minikube

configuran automáticamente según los valores predeterminados. Estos valores los indica *Minikube* al clúster durante el inicio mediante el archivo de configuración **kubeconfig**.

```
root@ubuntu:~# minikube start
 minikube v1.0.1 on linux (amd64)
 Downloading Minikube ISO ...
 142.88 MB / 142.88 MB [==========================================] 100.00% 0s
 Downloading Kubernetes v1.14.1 images in the background ...
 Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
 "minikube" IP address is 192.168.99.100
 Configuring Docker as the container runtime ...
 Version of container runtime is 18.06.3-ce
 Waiting for image downloads to complete ...
 Preparing Kubernetes environment ...
 Downloading kubeadm v1.14.1
 Downloading kubelet v1.14.1
 Pulling images required by Kubernetes v1.14.1 ...
 Launching Kubernetes v1.14.1 using kubeadm ...
 Waiting for pods: apiserver proxy etcd scheduler controller dns
 Configuring cluster permissions ...
 Verifying component health .....
 kubectl is now configured to use "minikube"
 Done! Thank you for using minikube!
```

Figura 22: Inicio de Minikube

Si se observa el archivo **kubeconfig** mediante el comando `kubectl config view` (Figura 23) se puede acceder a la configuración del clúster donde se puede comprobar la concordancia entre los valores mostrados al iniciar *Minikube* y del propio archivo como son la dirección IP del servidor o el nombre del clúster, entre otros muchos que aparecen.

```
root@ubuntu:~# kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/jf/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
```

Figura 23: Configuración de Minikube

4.4. ORQUESTACIÓN MULTICONTENEDOR: KOMPOSE

De igual manera que *Docker* ofrece facilidades para levantar múltiples contenedores simultáneos, en *Kubernetes* existe la posibilidad de crear todos los objetos necesarios en el clúster en un único documento siempre y cuando se especifique el tipo de objeto con sus campos correspondientes.

Este documento no tiene una sintaxis igual a la de un *Dockerfile* o un archivo *docker-compose* y dado que muchos desarrolladores comienzan el proyecto por el levantamiento de contenedores por ser la parte más básica, la traducción de *Docker* a objetos de *Kubernetes* puede ser una tarea difícil**.** Para ello, existe la posibilidad de fusionar ambas herramientas y crear de manera automática los objetos necesarios gracias a *Kompose*.

Kompose es una herramienta que ayuda a los usuarios familiarizados con *Docker-Compose* a convertir los archivos en recursos de *Kubernetes*[27].

[27] Kubernetes, Kompose. (2019). Recuperado de https://github.com/kubernetes/kompose

Su gran uso por parte de los desarrolladores es debido a la facilidad de conversión, ya que necesita únicamente tres comandos básicos y rapidez para poder disponer del despliegue originario en un clúster de producción.

Para su instalación, siguiendo el mismo método que hasta ahora, será necesario descargar el archivo binario, preferiblemente la última versión desde GitHub, hacerlo ejecutable y moverlo al *PATH*. Para realizarlo en Ubuntu los comandos a introducir son:

1. Descargar el archivo binario
   ```
   $ curl -L
   https://github.com/kubernetes/kompose/releases/download/v1.18.0
   /kompose-linux-amd64 -o kompose
   ```
2. Hacerlo ejecutable
   ```
   $ chmod +x kompose
   ```
3. Añadirlo al *PATH*
   ```
   $ sudo mv ./kompose /usr/local/bin/kompose
   ```

El uso de *Kompose* se basa en tres simples comandos[28]:

- `kompose convert`: Convierte los archivos de *Docker* en archivos de objetos de *Kubernetes*.
- `kompose up`: Convierte y despliega el archivo de *Docker* en un clúster de *Kubernetes*.
- `kompose down`: Deshace el despliegue generado eliminando los despliegues y servicios creados. Si se necesitan eliminar otros recursos generados durante la ejecución será necesario hacer uso de la API de *Kubernetes* **kubectl**.

Como función extra, *Komopose* también admite atributos, siendo el más utilizado a la hora de levantar los contenedores `--build none` el cual deshabilita la creación de la imagen y la subida de la misma al repositorio.

Para mostrar un ejemplo del uso de *Kompose* se han creado cuatro servicios iguales utilizando el mismo archivo y empaquetado para los cuatro. Todos los servicios siguen la misma estructura mostrada a continuación, cambiando únicamente el nombre y el puerto expuesto.

```
services:
  app1:
    image: app1
    container_name: app1-container
    network_mode: "host"
    expose:
      - "8080"
```

Haciendo uso de `kompose convert` se transforma el único archivo *docker-compose.yaml* en varios (en esta caso cuatro) archivos *deployment.yaml* como se muestra en la Figura 24.

[28] User Guide. (2019). Recuperado de http://kompose.io/user-guide/

```
jf@ubuntu:~/JoseF/Microservicios$ kompose convert
INFO Kubernetes file "app-deployment.yaml" created
INFO Kubernetes file "app1-deployment.yaml" created
INFO Kubernetes file "app2-deployment.yaml" created
INFO Kubernetes file "app3-deployment.yaml" created
```

Figura 24: Conversión de archivos mediante kompose

Como bien muestra la salida del programa, se han creado cuatro archivos con nombres diferentes que forman los cuatro servicios que existían en el archivo *docker-compose.yaml*

Para ejecutarlo en *Kubernetes* podría utilizarse la API con el comando `kubectl apply -f` o bien directamente con *kompose* mediante `kompose u` como se muestra en la Figura 25.

```
root@ubuntu:~/JoseF/Microservicios# kompose up --build none
INFO We are going to create Kubernetes Deployments, Services and PersistentVolu
meClaims for your Dockerized application. If you need different kind of resourc
es, use the 'kompose convert' and 'kubectl create -f' commands instead.

INFO Deploying application in "default" namespace
INFO Successfully created Deployment: app
INFO Successfully created Deployment: app1
INFO Successfully created Deployment: app2
INFO Successfully created Deployment: app3

Your application has been deployed to Kubernetes. You can run 'kubectl get depl
oyment,svc,pods,pvc' for details.
```

Figura 25: Ejecución de contenedores mediante kompose

De esta manera se crean los objetos necesarios dentro del clúster y se puede acceder a ellos mediante los distintos comandos que ofrece *Kubernetes*.

Por último, para terminar la ejecución se cierra con el comando `kompose down` (Figura 26) y la salida por pantalla informa que la operación se ha realizado correctamente.

```
root@ubuntu:~/JoseF/Microservicios# kompose down
INFO Deleting application in "default" namespace
INFO Successfully deleted Deployment: app
INFO Successfully deleted Deployment: app1
INFO Successfully deleted Deployment: app2
INFO Successfully deleted Deployment: app3
```

Figura 26: Eliminación de los contenedores mediante kompose

4.5. RED DE KUBERNETES

La red es la parte central de *Kubernetes* ya que basa todo su funcionamiento en compartir máquinas entre aplicaciones. Normalmente al compartir máquinas se requiere asegurar que dos aplicaciones no utilizan el mismo puerto pero coordinar estos puertos no es una tarea simple cuando se realizan escalados de la aplicación y pone en peligro el correcto funcionamiento del clúster[29].

La asignación dinámica de puertos conlleva distintos problemas ya que todos los objetos que se comunican o quieran comunicarse con una aplicación en concreto deben conocer en todo momento

[29] Cluster Networking. (2019). Recuperado de https://kubernetes.io/docs/concepts/cluster-administration/networking/

los cambios que se producen en cuanto a la red se refiere. Es por ello que *Kubernetes* tiene un enfoque diferente para esto.

En el modelo de red que ofrece *Kubernetes* cada *pod* tiene su propia dirección IP, por lo que no es necesario crear enlaces explícitos entre *pods*, el enlace existe mediante la IP, y tampoco es necesario el mapeo (excepto en situaciones muy concretas) entre los puertos de los contenedores y del host, ya que la comunicación se hace mediante el *pod*. Esto crea un modelo donde los *pods* pueden ser considerados como máquinas virtuales desde la perspectiva de la asignación de puertos, nombres, descubrimiento de servicios, balanceado de carga... en definitiva, de todas las actividades referentes a la red.

El modelo de red de *Kubernetes* tiene dos reglas fundamentales por lo que, siempre que se cumplan, puede implementarse un modelo distinto. Estas reglas son[30]:

- Todos los contenedores tienen que ser accesibles entre ellos sin necesidad de NAT sin importar en el nodo que se encuentren.
- Todos los nodos deben poder comunicarse con todos los contenedores.

La comunicación dentro del clúster puede ser a diferentes niveles: contenedor-contenedor, *pod-pod* (mismo y diferente nodo), o *pod*-servicio.

- Comunicación **contenedor-a-contenedor**: Cada *pod* en *Kubernetes* tiene su propia dirección IP mientras que los contenedores dentro del *pod* comparten el *namespace* de la red, por lo que se ven entre ellos como **localhost**. Este mecanismo es implementado por defecto por el contenedor encargado de crear la red, que actúa como *bridge* para dirigir el tráfico entre contenedores de un mismo *pod*.

- Comunicación **pod-a-pod**: Las direcciones IP de cada *pod* son accesibles desde cualquier otro *pod* pero la manera en que lo haga será diferente según si se encuentran en el mismo nodo o no.

  - Estando en el mismo nodo: Al encontrarse ambos nodos en el mismo *namespace* de la red la comunicación es directa mediante el *bridge* por defecto.
  - Estando en diferentes nodos: En este caso *Kubernetes* delega la implementación a la interfaz de red de contenedor (CNI por sus siglas en inglés *Container Network Interface).* Esta implementación puede ser elegida por el usuario utilizando el plugin de interfaz de red que desee. Cuando la información sale de un *pod* dirección a otro es el momento en que actúa el plugin elegido, mientras que en el interior de los nodos se sigue el proceso definido por *Kubernetes***.**

- Comunicación **pod-a-servicio**: *Kubernetes* es dinámico, los *pods* se crean y se destruyen constantemente. El concepto de servicio a rasgos generales es ser un intermediario entre el cliente y los *pods* por lo que lo normal es hacer uso de ellos en lugar de acceder al *pod* directamente. Al crear un servicio se crea un **endpoint** que indica la dirección IP del *pod* asociado, mientras que, por defecto, *Kubernetes* hace uso de **iptables** para conocer el destino, es decir, el *pod* que se encuentra tras el servicio.

[30] Saito, H., Lee, H., & Wu, C. (2017). DevOps with Kubernetes. Birmingham, Reino Unido: Packt Publishing Ltd.

CAPÍTULO 5
MONITORIZACIÓN

5. MONITORIZACIÓN

La monitorización se basa principalmente en la instalación de un software ya sea independiente o que forma parte de otro software, dedicado a la seguridad y vigilancia tanto para un sistema individual como en una red haciendo uso de reglas, preferencias o acciones específicas predefinidas que permiten describir el estado tanto normal como anormal del sistema[31].

En el ámbito del desarrollo y mantenimiento de software de cualquier sistema informático la monitorización y el registro de logs son una parte crucial, más aún si se trata de sistemas y aplicaciones distribuidas ya que debido al mayor número de máquinas la supervisión se complica de forma exponencial al ser necesario realizar distintas tareas para cada una de ellas. No es estrictamente necesario seguir un protocolo formal para la depuración de un proyecto, pero sí hay acciones básicas necesarias para llevarla a cabo como acceder a toda la información necesaria proveniente de distintas fuentes, recopilar y almacenar los registros de cada uno de los procesos existentes para cada contenedor y visualizar dicha información para poder encontrar cualquier posible error de la manera más rápida posible.

Hoy en día cualquier aplicación necesita estar en ejecución en todo momento lo que hace que cualquier caída suponga una situación crítica para la empresa. Por esta razón la monitorización se hace indispensable en cualquier proyecto, lo que ha significado un aumento de la demanda respecto a este tipo de software y por tanto un aumento considerable del número de proveedores de monitorización en los últimos años, pudiendo encontrar en el mercado soluciones e integraciones especializadas en cualquier ámbito de la informática como el uso de contenedores.

5.1. MONITORZACIÓN DE CONTENEDORES

Para la monitorización en contenedores la propia herramienta *Docker* proporciona comandos con los que se puede acceder a *logs* o registros tanto de un contenedor como de la imagen mediante la que se creó.

Con el comando `docker inspect` se obtiene toda la información sobre un contenedor como por ejemplo cuándo se creó, qué comandos se pasaron por defecto al contenedor, qué puertos tiene mapeados o cuál es la dirección IP entre muchas otras opciones.

El comando `docker inspect` admite un parámetro de entrada para especificar los valores que se quieren obtener con el uso de `-f (--format)`. Así el uso básico del comando para mostrar información de un contenedor cualquiera sería:

```
$ docker inspect <nombre_contenedor>
```

Si utilizamos el comando sobre un contenedor recién creado, por ejemplo, un contenedor que contenga una imagen de *Tomcat* con nombre “contenedor-ejemplo” se obtiene una salida similar a la mostrada en la Figura 27.

[31] What is Monitoring Software?. (2019). Recuperado de https://www.techopedia.com/definition/4313/monitoring-software

```
jf@ubuntu:~$ docker inspect contenedor-ejemplo
[
    {
        "Id": "8241ad2c18557acbf44223ba0dc4c19ce6a639b482e9062af0b432c93b275c63",
        "Created": "2019-05-21T14:33:36.236273537Z",
        "Path": "catalina.sh",
        "Args": [
            "run"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 6094,
            "ExitCode": 0,
            "Error": "",
```

Figura 27: Inspección de un contenedor

Se muestran unos pocos parámetros de todos los totales, cada cual indica una propiedad del contenedor y cada campo tiene su propia utilidad. Por ejemplo, en este caso en concreto con este comando, podríamos comprobar que el contenedor, en el momento de insertar el comando, se encuentra en ejecución sin ningún error.

Para mostrar unos valores específicos mediante la opción `–f` se deben indicar los atributos del formato JSON devuelto anteriormente, por ejemplo, para obtener la dirección IP el comando correcto sería:

```
$ docker inspect --format=

'{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
<NOMBRE_CONTENEDOR>
```

Esto es debido a la salida en formato JSON que proporciona *Docker*. En este caso **IPAddress** es un atributo de la lista **Networks** que a su vez lo es de **NetworkSettings** como se muestra en la Figura 28.

```
$ docker inspect --format='{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
...
 "NetworkSettings": {
           ...
           "Networks": {
                   ...
                   "IPAddress": "172.17.0.2",
                   ...
```

Figura 28: Estructura JSON

De esta manera se puede acceder a cualquier atributo que se muestre en la salida original.

Ejecutando el comando aparecerá la IP mediante la cual acceder al contenedor, en la Figura 29 además se ha hecho *curl* a dicha IP para comprobar que efectivamente en el puerto 8080 se encuentra *Tomcat* en ejecución.

```
jf@ubuntu:~$ docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' contenedor-ejemplo
172.17.0.2
jf@ubuntu:~$ curl 172.17.0.2:8080


<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Apache Tomcat/8.5.41</title>
        <link href="favicon.ico" rel="icon" type="image/x-icon" />
        <link href="favicon.ico" rel="shortcut icon" type="image/x-icon" />
        <link href="tomcat.css" rel="stylesheet" type="text/css" />
    </head>
```

Figura 29: Inspección de un atributo específico

Otro comando útil provisto por *Docker* es `docker stats` que muestra las estadísticas referentes al uso de recursos de un contenedor específico.

Este comando muestra las estadísticas de un contenedor en ejecución, es por eso que la salida que muestra se actualiza constantemente con el uso actual de cada momento. Su uso es mediante el comando:

```
$ docker stats <nombre_contenedor>
```

Y la salida similar a la mostrada en la Figura 30.

```
CONTAINER ID   NAME                 CPU %    MEM USAGE / LIMIT     MEM %    NET I/O           BLOCK I/O     PIDS
8241ad2c1855   contenedor-ejemplo   0.15%    127.5MiB / 3.83GiB    3.25%    26.3kB / 105kB    2.47MB / 0B   48
```

Figura 30: Estadísticas de un contenedor

Para la monitorización de los eventos ocurridos en el ciclo de vida de los contenedores como por ejemplo la creación, parada o destrucción de los mismos, el comando que provee *Docker* es `docker events` que permite distintos parámetros de entrada para seleccionar un rango de tiempo específico.

Mediante este comando se obtienen eventos en tiempo real desde el servidor de *Docker.* El comando no accede a los contenedores para devolver los datos sino que utiliza el registro del *host* de *Docker* donde se encuentra la información relativa al ciclo de vida de los contenedores que gestiona.

Se puede utilizar el comando de la manera:

```
$ docker events <opción>
```

Se puede observar en el siguiente ejemplo el uso del parámetro `--since` para establecer un periodo de tiempo máximo. Como ejemplo en la Figura 31 se muestran todos los eventos ocurridos en *Docker* desde el día introducido por consola, que corresponden con la descarga de la imagen de *Tomcat* y creación del contenedor sobre el que se han basado todos los ejemplos anteriores expuestos.

```
jf@ubuntu:~$ docker events --since 2019-05-21
2019-05-21T07:30:16.207759458-07:00 image pull tomcat:latest (name=tomcat)
2019-05-21T07:30:16.414136569-07:00 container create 612f88e64fe148ae87805f05a
2019-05-21T07:30:16.420354458-07:00 container attach 612f88e64fe148ae87805f05a
2019-05-21T07:30:16.464408097-07:00 network connect 3523b9c87dd8e8d82a987d897d
```

Figura 31: Eventos en los contenedores en un lapso de tiempo

5.2. MONITORIZACIÓN DEL CLÚSTER

La monitorización de los contenedores permite conocer el estado de cada uno de ellos de forma individual, pero a la hora de crear un clúster surge el problema de tener múltiples contenedores por lo que la revisión de todos ellos puede suponer una tarea repetitiva, tediosa y propensa a errores. Dentro de un clúster de *Kubernetes* pueden estar ejecutándose múltiples objetos de manera concurrente; un único *namespace* con un servicio incluye, como mínimo, que haya un *pod* ejecutando un despliegue con un contenedor, por lo cual, con la presencia de tantos objetos, que además pueden ir variando en el tiempo (p, ej. reescalado), encontrar dónde se produce un error cuando falla el clúster se hace una tarea imposible. Además, debido a la naturaleza efímera de los contenedores es posible que desde que se produzca el fallo hasta el momento de realizar la depuración, el contenedor haya desaparecido, lo que convierte a los archivos de *logs* en una herramienta fundamental e indispensable.

Pese a todo esto, Kubernetes tiene una gran capacidad para recuperarse de fallos de manera automática, como reiniciar un *pod* o balancear la carga en los nodos, pero en ocasiones no es suficiente y es necesario realizar el proceso de forma manual; para estos casos es necesario monitorizar la ejecución del clúster haciendo uso de distintas herramientas, desde las propias de *Kubernetes* como el panel de control hasta software externos especializados.

Para monitorizar un proceso hay dos factores principales a tener en cuenta: **Qué** se puede monitorizar y **cómo** hacerlo.

En cuanto al ***qué*** monitorizar, *Kubernetes* por sí mismo ofrece la posibilidad de conocer:

- Uso de CPU: La monitorización revela el uso de CPU del sistema y del usuario además de esperas de lectura y escritura. Es útil para encontrar cuellos de botella en el despliegue.
- Uso de memoria: Muestra la cantidad de memoria disponible y en uso tanto para la memoria libre como para la caché.
- Uso del disco: Indica el espacio en disco. La falta de espacio en disco puede ocasionar un fallo en la ejecución de un programa, por lo que es necesario vigilarlo.
- Ancho de banda de red: Ofrece el ancho de banda en uso y disponible. Aunque hoy en día parezca imposible llegar a consumir el ancho de banda es importante vigilarlo por si se producen comportamientos sospechosos como puede ser un ataque *DDoS.*
- Recursos de los *Pods*: Se pueden acceder a los distintos recursos que está utilizando un *Pod* en concreto pudiendo ser esta información utilizada por el *scheduler* (planificador) de y colocar los *pods* en nodos donde haya recursos disponibles (auto escalado).

Respecto al ***cómo*** hacerlo, para las métricas mencionadas anteriormente la manera más sencilla es haciendo uso del panel de control de *Kubernetes*.

Sin embargo, los recursos disponibles para la monitorización se quedan cortas para todas las posibilidades existentes, es por ello que *Kubernetes* permite hacer uso de métricas personalizadas. Es aquí donde vuelven a surgir las preguntas ¿**qué** métricas personalizadas puede leer el clúster? y ¿**cómo** las lee?

Las métricas disponibles son todas aquellas que estén escritas de manera que la API de *Kubernetes* pueda acceder a ella y para su lectura se debe hacer uso de programas externos. Como se menciona anteriormente, el más utilizado por la comunidad es *Prometheus*.

5.2.1. KUBERNETES DASHBOARD

Además de la línea de comandos, *Kubernetes* proporciona una interfaz web de usuario donde ver el estado del clúster, así como interactuar con él. Se puede utilizar tanto para implementar aplicaciones en un clúster de *Kubernetes* como para solucionar problemas o administrar los recursos existentes. La interfaz muestra una descripción general del estado actual del clúster en tiempo real además de todos

los objetos que lo componen, pudiendo interaccionar con ellos por lo que, por ejemplo, se puede escalar un servicio o reiniciar un *pod*[32].

Para hacer uso de ella es necesario iniciarla mediante el comando:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/deploy/recommended/kubernetes-dashboard.yaml
```

Este comando creará un *pod* a partir de las especificaciones oficiales del repositorio de *Kubernetes* en *Github*.

Si se trabaja con *Minikube* únicamente hay que activarla como *addon*. Para listar los *addons* se utiliza el comando `$ minikube addons list`

La lista será similar a la mostrada en la Figura 32 donde "*dashboard"* aparece inactiva. Con el comando `minikube addons enable <addon>` se activa y se hace accesible mientras que se utiliza `minikube addons disable <addon>` para desactivarlo y eliminarlo del clúster.

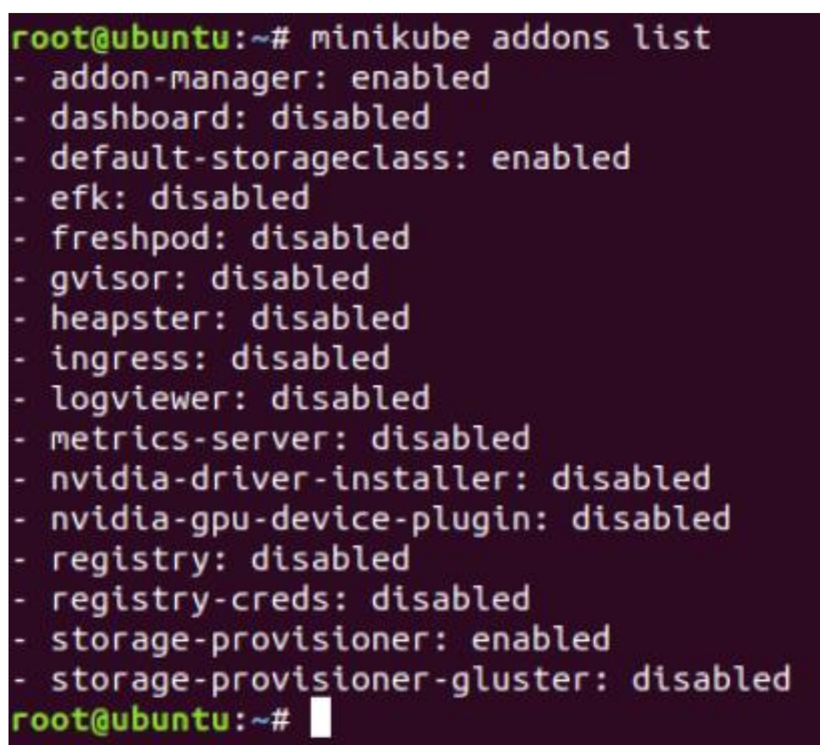

Figura 32: Lista de addons disponibles en Minikube

Para acceder a ella desde *Minikube* la manera más simple es hacer uso del comando `minikube dashboard` y se abrirá en una ventana del navegador predeterminado.

Mientras que si no se trabaja en *Minikube* para acceder a la interfaz es necesario primero crear un proxy entre el terminal y la API de *Kubernetes* mediante el comando **kubectl proxy**. Una vez que el proxy esté levantado se puede acceder al puerto abierto que por defecto es **8001** y estará mapeado al motor de *Kubernetes.* Por lo que accediendo a la dirección http://localhost:8001/ui aparecerá una ventana similar a la mostrada en la Figura 33 si el clúster no se encuentra vacío.

[32] Web UI (Dashboard). (2019). Recuperado de https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

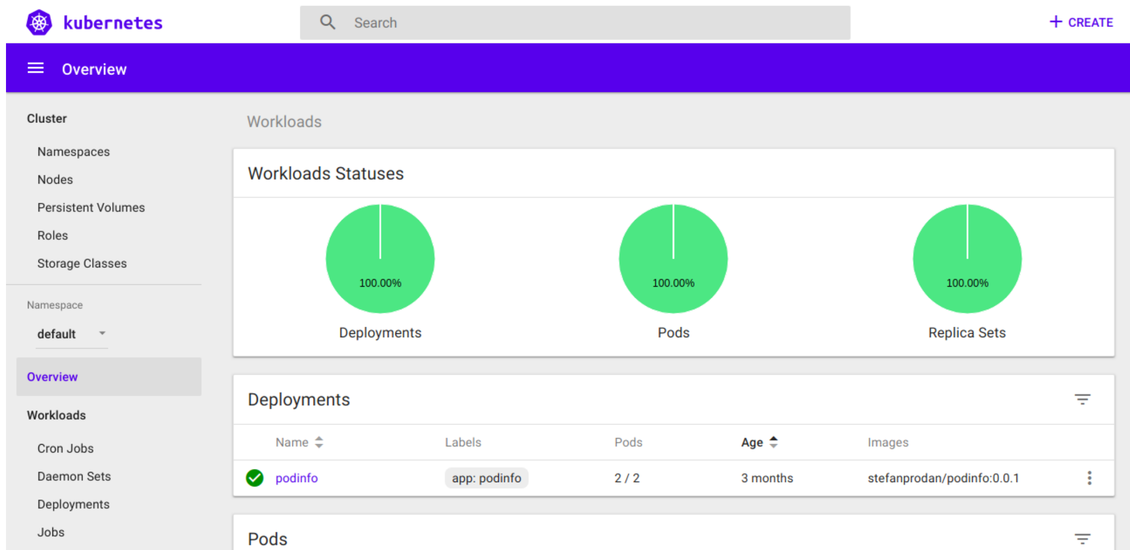

Figura 33: Dashboard de Kubernetes

La interfaz permite múltiples opciones para elegir, tanto *pods* como nodos, volúmenes o *namespaces*. Pero la parte que nos interesa para la monitorización es, en primer lugar, la visualización del uso de memoria y CPU de los despliegues dentro del clúster. Aunque para ello es necesario activar otro *addon*, en este caso *Heapster* y darle permisos dentro del clúster. Como se muestra en la Figura 34, con esta configuración se puede conocer el gasto que está teniendo cada uno de ellos.

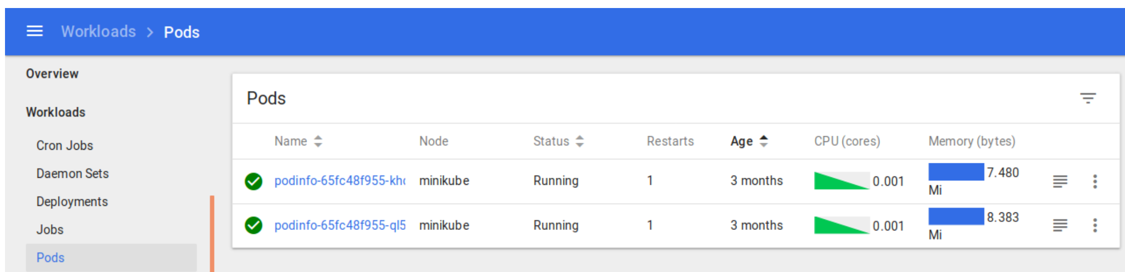

Figura 34: Lectura de consumo de recursos mostrada en la interfaz web de Kubernetes

Así como los *logs* que se muestran en la Figura 35 o cualquier otra opción a ejecutar en cada contenedor.

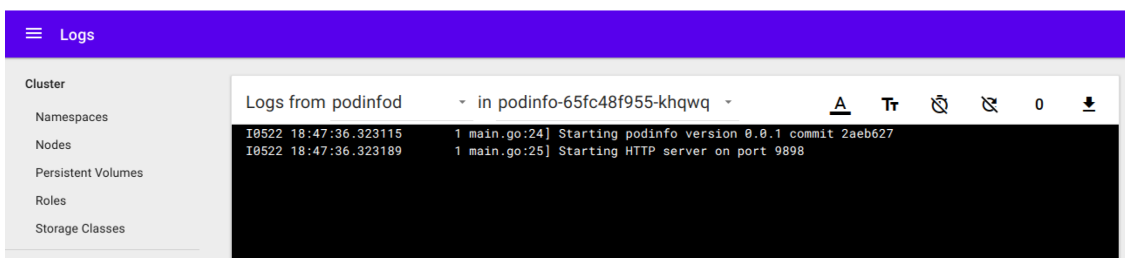

Figura 35: Logs de un contenedor desde la interfaz web de kubernetes

Si bien esta interfaz no nos añade ninguna nueva funcionalidad que no estuviese disponible mediante comandos sí que facilita el uso y, sobre todo, el visionado de los eventos que ocurren en el interior del contenedor. Para obtener información adicional es necesario hacer uso de un software que lea los

valores del clúster. Los componentes de *Kubernetes* exponen las métricas en un formato en concreto, por ello haciendo uso de ese mismo formato se pueden utilizar herramientas que ayuden a la monitorización del clúster. La herramienta más popular en este ámbito es *Prometheus*.

5.2.2. PROMETHEUS

Prometheus es un conjunto de herramientas de monitorización y alerta de código abierto. Fue desarrollado en 2012 por la empresa SoundCloud, pero más adelante pasaría a ser un proyecto de código abierto, uniéndose en 2016 a la *Cloud Native Computing Foundation* siendo el segundo proyecto acogido tras *Kubernetes* [33].

Al igual que cualquier otro software de monitorización, *Prometheus* se basa en agentes que extraen estadísticas de los componentes del sistema, indicados a la izquierda en la Figura 36 [34]. *Prometheus* dispone de las métricas constantemente de forma activa, es decir, se encarga de leer en todo momento los datos requeridos en lugar de esperar una respuesta que puede no acoplarse a los tiempos de espera estimados por el usuario. Además, el software puede enviar alertas según las reglas preconfiguradas al manager ("*alertmanager*"). Este manager se encarga de administrar las alarmas, agrupando las recibidas y enviando a otra aplicación encargada de transmitir el mensaje. También es posible hacer uso de otro software que permita visualizar todos los datos leídos como puede ser la propia interfaz de usuario de *Kubernetes* [35].

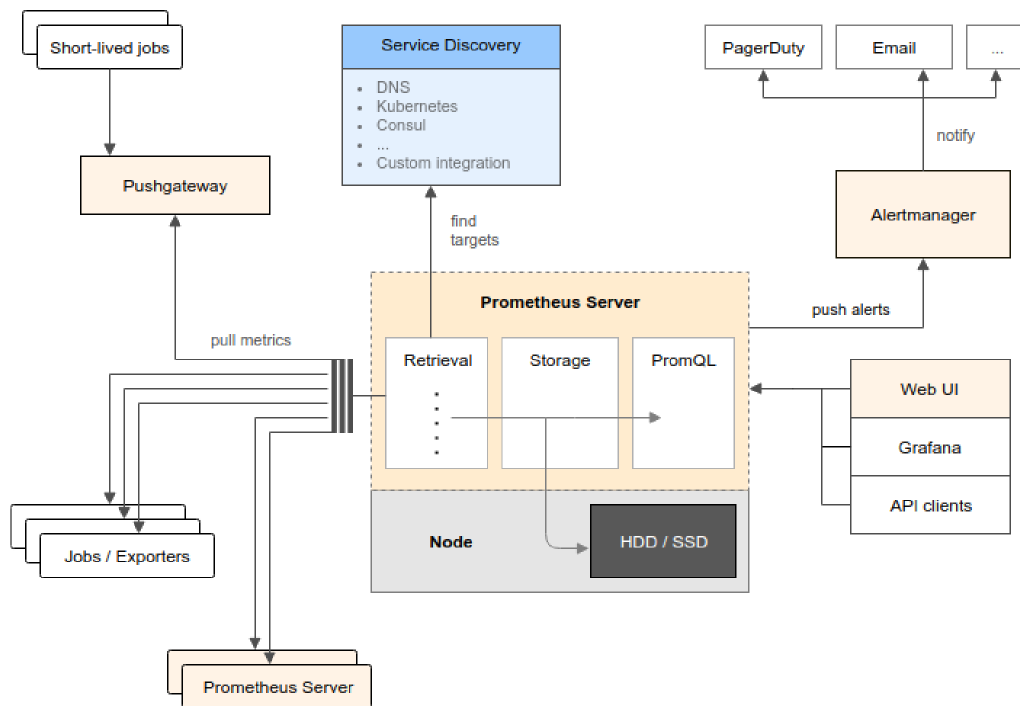

Figura 36: Arquitectura de Prometheus.

[33] Overview | Prometheus. (2019). Recuperado de https://prometheus.io/docs/introduction/overview/
[34] Prometheus – Architecture overview. (2019). Recuperado de https://github.com/prometheus/prometheus
[35] Saito, H., Lee, H., & Wu, C. (2017). DevOps with Kubernetes. Birmingham, Reino Unido: Packt Publishing Ltd.

Prometheus permite dos tipos de reglas posibles para configurar y ser evaluadas en intervalos de tiempo predefinidos, estas son:

- Reglas de grabación, que permite ejecutar acciones que se requieren en repetidas ocasiones o computacionalmente costosas y guardar el resultado como uno nuevo.

- Reglas de alerta, que permiten definir condiciones bajo las cuales el programa envía una notificación. Estas alertas tienen que ser escritas en el lenguaje de *Prometheus* para que éste las entienda y pueda ejecutarlas.

El software contiene una base de datos local en disco para almacenar los datos correspondientes, pero también puede utilizarse en sistemas remotos[36].

Son varias las funcionalidades que forman *Prometheus*, siendo las más destacables, como se muestra en la Figura 37:

- **Alertmanager** que gestiona las alertas enviadas por las aplicaciones o el propio servidor de *Prometheus*[37].
- **Prometheus operator** proporciona definiciones de monitoreo a los servicios de *Kubernetes* y al despliegue de *Prometheus* haciendo que la configuración dentro del clúster sea nativa administrando las instancias necesarias[38].

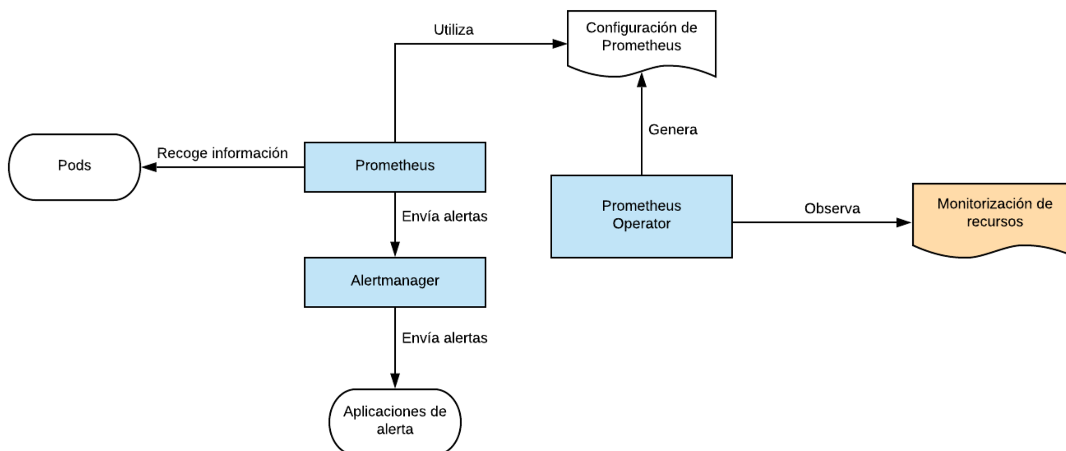

Figura 37: Funcionamiento de Prometheus

5.2.2.1. DESPLEGAR PROMETHEUS EN EL CLÚSTER

Una vez conocido el funcionamiento básico de *Prometheus* es necesario instalarlo en el clúster. Ya que este software es genérico para la monitorización se puede configurar para amoldarse a los requisitos especificados de diferentes maneras. A continuación, se hará uso siempre que sea posible de archivos de configuraciones oficiales de *Prometheus* ya que, al estar algunas de sus funcionalidades en fase beta, conviene seguir los pasos que los creadores acreditan que funcionan correctamente.

[36] Storage | Prometheus. (2019). Recuperado de https://prometheus.io/docs/prometheus/latest/storage/
[37] Alertmanager | Prometheus. (2019). Recuperado de https://prometheus.io/docs/alerting/alertmanager/
[38] Prometheus Operator (2019). Recuperado de https://github.com/coreos/prometheus-operator

Para poder acceder a *Prometheus* es necesario desplegarlo como cualquier otro objeto del clúster, es decir, será necesario crear un despliegue que contenga un contenedor que ejecuta una imagen. Dicho despliegue será gestionado por un *pod* y para acceder a él se hará uso de un servicio.

Antes de empezar conviene crear un *namespace* dentro del clúster sobre el que desplegar todos los objetos necesarios. Esta acción no es obligatoria pero sí ayuda a tener buenos hábitos y una estructura más clara del clúster. Este *namespace* lo llamaremos "monitoring" ya que es el uso más recurrente por la comunidad de usuarios de *Kubernetes*.

Antes de desplegar los contenedores hay que configurar *Prometheus*, para ello hay dos opciones. La que mejor cumple con los estándares de uso de *Docker* es configurar una imagen con la que levantar el contenedor y si es necesario una actualización realizarla sobre la imagen y volver a relanzar el contenedor con la nueva actualización. La otra opción es levantar el contenedor con una imagen con configuración estándar y asociarle un objeto **ConfigMap** y en caso de querer actualizar la configuración únicamente es necesario modificar el archivo. Ambas soluciones son correctas y funcionales y la elección de ambas varía mínimamente a la hora de ejecutar *Prometheus*. Sea cual sea la opción elegida hay que crear una configuración y luego decidir si asociarle un **ConfigMap** o no. Se puede encontrar una plantilla de dicha configuración en el repositorio oficial de *Prometheus* (https://github.com/prometheus/prometheus) con los campos mínimos necesarios y en caso de querer mapear la configuración con un objeto de *Kubernetes* únicamente sería necesario añadir el contenido en un archivo *.yml* de la siguiente manera:

```
apiVersion: v1
data:
  prometheus.yml: |
  #
  # Contenido del archivo de configuración de Prometheus
  #
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: monitoring
```

De esta manera se añade toda la configuración inicial a un objeto **ConfigMap** de nombre *prometheus-config* que se encuentra en el *namespace* creado anteriormente *monitoring*.

Con la configuración necesaria se puede desplegar el contenedor que incluye la imagen de *Prometheus*. Para ello es necesario crear diversos objetos en el clúster: un rol (**ClusterRole**) para tener distintos permisos (*get, list…*) en el clúster, un **ServiceAccount** para identificar el proceso con el nombre de *Prometheus* y un **ClusterRoleBinding** que asigne el rol creado al proceso indicado, teniendo estos objetos un esquema similar al mostrado a continuación:

```
apiVersion: . . .
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
  - . . .
  - services
  - pods
  - . . .
  verbs:["get"]
- apiGroups:
  - extensions
  resources:
  - . . .
```

```
apiVersion: . . .
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: monitoring
```

```
apiVersion: . . .
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: . . .
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: prometheus
  namespace: monitoring
```

El `ClusterRole` de nombre *Prometheus* tiene la opción de acceder (*get*) a los recursos de los servicios y *pods* mientras que el `ServiceAccount` identifica al proceso con el nombre de `prometheus` para que el `ClusterRoleBinding` pueda asignarle los permisos del rol creado

Configurados los permisos para el proceso que se quiere ejecutar, llega el turno de levantar el contenedor con dicho proceso. Para ello se hace uso de una plantilla con un despliegue en el clúster que utilice la imagen oficial con la configuración especificada. Para el correcto funcionamiento según la configuración del clúster es necesario añadir unas modificaciones a la plantilla oficial, que son:

- Añadir el **ConfigMap** en la especificación de los contenedores
- Nombrar el despliegue como "prometheus" para poder enlazarlo con un servicio más adelante
- Añadir la anotación: `prometheus.io/scrape: "true"`

Además, opcionalmente se puede añadir un volumen persistente, ya que por defecto el volumen creado es **emptyDir** por lo que si se reinicia el contenedor se pierden todos los datos recogidos hasta el momento.

Con estos cambios el archivo para crear el despliegue quedaría con una estructura similar a la mostrada a continuación:

```
apiVersion: . . .
kind: Deployment
metadata:
  name: prometheus
  namespace: monitoring
spec:
  selector:
    matchLabels:
      name: prometheus #Etiqueta para el servicio
    . . .
  template:
    metadata:
      . . .
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "9090" #Puerto para el servicio
    spec:
      containers:
        image: quay.io/prometheus/prometheus #oficial
      . . .
      volumes:
      - emptyDir: {}
        name: data
      - configMap:
          name: prometheus-config
        name: config-volume
```

Consiguiendo así crear un despliegue con el nombre de `prometheus` en el espacio de nombres `monitoring` con la imagen provista por la organización y haciendo uso de la configuración creada anteriormente y almacenada en el objeto `config-volume`.

Con el despliegue en funcionamiento para poder acceder a la interfaz gráfica es necesario crear un servicio en el *namespace* correspondiente al puerto indicado en el archivo anterior (9090), siendo el archivo resultante:

```
apiVersion: v1
kind: Service
metadata:
  name: prometheus
  namespace: monitoring
spec:
  ports:
  - port: 9090
    protocol: TCP
    targetPort: 9090
  selector:
    name: prometheus
  type: NodePort
```

Toda la configuración es coherente con la anterior para que pueda haber concordancia, el puerto, nombre o *namespace* debe coincidir con los objetos creados. Además, el servicio se crea de tipo `NodePort` para que esté accesible a todos los nodos del clúster. Todo este proceso se muestra en la Figura 38.

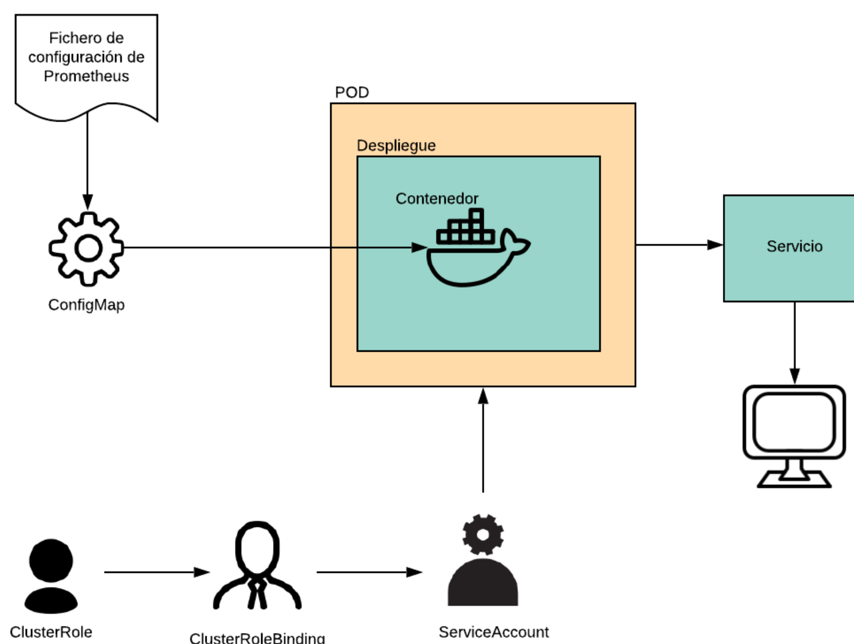

Figura 38: Representación de la configuración del operador de Prometheus

Donde por un lado se crea la configuración a través de un archivo existente para posteriormente insertarla en el contenedor mientras por otro se crea un rol que permite al proceso la lectura de datos y se le asigna al *pod* creado. Configurado el despliegue se expone el *pod* a través de un servicio para poder acceder desde el navegador.

Al acceder a la IP del servicio creado, si la configuración y el despliegue han sido correctos debería mostrarse la interfaz de *Prometheus*, similar a la que aparece en la Figura 39.

Para ello primero se consigue la IP mediante el comando `$ minikube service prometheus -n monitoring --url` y se accede a la dirección mostrada por pantalla.

Como dato extra, el uso de *Prometheus* es tan recurrente que existe actualmente la idea de añadirlo como *addon* a *Kubernetes*. Actualmente se encuentra en fase experimental y todavía no está disponible. Para más información el repositorio donde se desarrolla el proyecto es el siguiente:

https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/prometheus

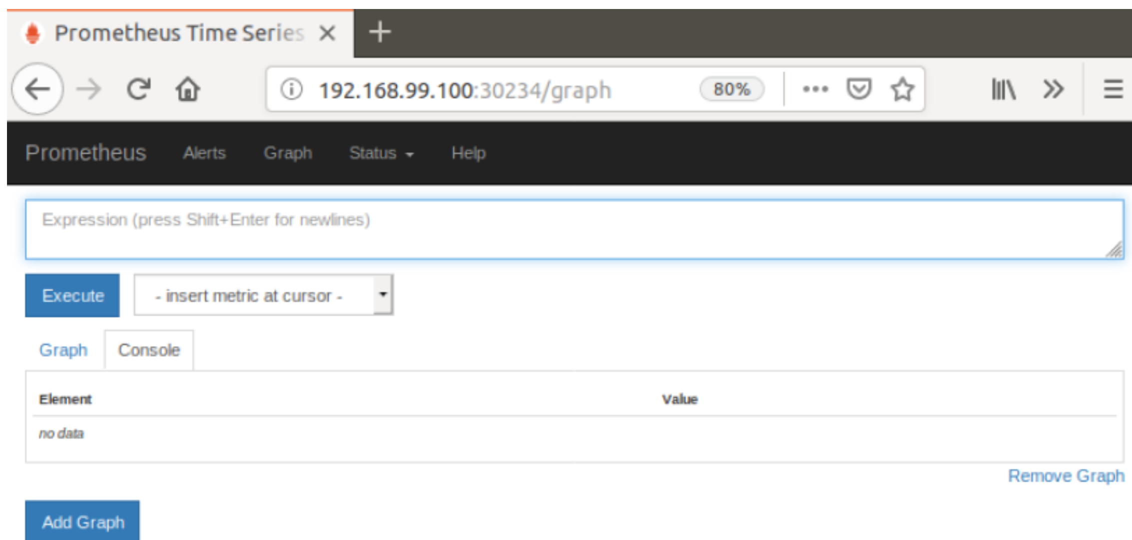

Figura 39: Interfaz web de Prometheus

CAPÍTULO 6

ESCALADO DEL CLÚSTER

6. ESCALADO DEL CLÚSTER

Normalmente no todas las aplicaciones ejecutadas en un clúster se ven sometidas a la misma carga de trabajo. Podemos encontrar aplicaciones con peticiones constantes y otras que son solicitadas de manera excepcional. Todas ellas deben convivir en un mismo clúster y aunque tengan mayor o menor uso todas son imprescindibles para el correcto funcionamiento de la aplicación que componen.

Para ofrecer el mejor servicio al usuario, el clúster debe estar operativo y accesible en todo momento que se requiera con la menor latencia posible. Si el proceso principal recibe mucha carga y ralentiza toda la aplicación, el producto pierde valor. Para evitar estas situaciones la mejor solución es duplicar -escalar- los procesos necesarios. Con la llegada de la arquitectura de microservicios este escalado es más simple ya que no es necesario realizarlo sobre toda la aplicación sino sobre las partes que lo requieran; además con la encapsulación en contenedores se facilita aún más el trabajo ya que únicamente es necesario duplicar el contenedor. *Kubernetes* facilita el escalado ya que dispone de comandos específicos para realizarlo. El comando en cuestión es **scale**.

El comando **scale** acepta diferentes definiciones, siendo la más usual:

```
kubectl scale --replicas=<NUM_REPLICAS> <NOMBRE_RECURSO>
```

Como con otros objetos, la API de *Kubernetes* permite indicar los parámetros en un archivo *.yaml* haciendo uso de **-f**:

```
kubectl scale --replicas=<NUM_REPLICAS> -f <RUTA_ARCHIVO>
```

Estos comandos escalarán los objetos indicados al número de réplicas indicado. Si la cantidad total es menor de la indicada se añadirán tantas como sea necesario y si la cantidad es menor se eliminarán contenedores hasta coincidir en número.

Existe un parámetro opcional `--current-replicas` como verificación previa que solo realiza el escalado en caso de que las réplicas actuales coincidan con las indicadas.

El escalado se realiza de la siguiente manera:

Se tiene un despliegue cualquiera, en este caso llamado *hello-node* y, como se indica en la salida por pantalla en la Figura 40, con un único *pod* con nombre compuesto por el propio nombre del despliegue más una serie de caracteres aleatorios, mostrado en la Figura 41.

```
root@ubuntu:~# kubectl get deployment
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
hello-node   1/1     1            1           3m48s
```

Figura 40: Despliegues existentes en el clúster

```
root@ubuntu:~# kubectl get po
NAME                          READY   STATUS    RESTARTS   AGE
hello-node-78cd77d68f-s7tlg   1/1     Running   0          3m7s
```

Figura 41: Pods existentes en el clúster

Al realizar un escalado a, por ejemplo, tres réplicas mediante el comando de la figura 42 el resultado es la creación de dos nuevos *pods* pero no de nuevos despliegues, ya que sigue siendo uno único con múltiples "copias".

```
root@ubuntu:~# kubectl scale --replicas=3 deployment/hello-node
deployment.extensions/hello-node scaled
```

Figura 42: Escalado mediante línea de comandos

Ejecutando los mismos comandos anteriores para visualizar los *pods* y despliegues del clúster vemos como ahora hay un único despliegue (Figura 43) y tres *pods* (Figura 44)

```
root@ubuntu:~# kubectl get deployment
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
hello-node   3/3     3            3           14m
```

Figura 43: Despliegues existentes tras el escalado

```
root@ubuntu:~# kubectl get po
NAME                          READY   STATUS    RESTARTS   AGE
hello-node-78cd77d68f-4ztrs   1/1     Running   0          42s
hello-node-78cd77d68f-rjchl   1/1     Running   0          42s
hello-node-78cd77d68f-s7tlg   1/1     Running   0          14m
```

Figura 44: Pods existentes tras el escalado

El escalado de las aplicaciones es útil en una gran parte de los posibles escenarios, pero no en todos, ya que en ocasiones la disponibilidad de la aplicación no depende únicamente de la correcta ejecución de las aplicaciones. Podría ser posible necesitar una actualización y los pasos que conlleva: detener el proceso, aplicar los cambios, volver a lanzarlo y esperar la puesta en marcha, como mínimo. Las actualizaciones pueden ocurrir cada mes, cada semana o incluso cada día. Sea cual sea el periodo de tiempo necesario, la parada de la aplicación es obligatoria. Para evitar esto, *Kubernetes* proporciona una opción denominada actualización progresiva (*rolling-update*) que permite actualizar distintos objetos en ejecución sin detenerlos siguiendo los siguientes pasos[39]:

1. Se crea un nuevo replicado del contenedor actualizado con la nueva configuración. Este replicado constará de tantos contenedores como sea necesario.
2. Se aumentan las réplicas nuevas mientras que se disminuyen las antiguas hasta alcanzar el número deseado de todas ellas.
3. Se elimina el contenedor original una vez que los replicados realizan el trabajo.

Para actualizar la imagen del contenedor se hace uso del siguiente comando:

```
kubectl set image <NOMBRE> <IMAGEN>=<IMAGEN_NUEVA:TAG>
```

Como ejemplo se ha actualizado la imagen anterior por la última de *openjdk* (Figura 45).

```
root@ubuntu:~# kubectl set image deployment/hello-node  hello-node=openjdk:latest
deployment.extensions/hello-node image updated
```

Figura 45: Actualización de la imagen de un contenedor

Esta manera de escalar aplicaciones es útil pero no es suficiente ya que todo el proceso es manual. Debe haber una persona que se encargue de observar el funcionamiento y, en caso de ser necesario, ejecutar los comandos convenientes para escalar las aplicaciones en función del uso que tenga en un momento dado. Para una mayor eficiencia este proceso debe ser automático, es por eso que *Kubernetes* proporciona una herramienta que permite escalar las aplicaciones de forma automática.

---

[39] Perform Rolling Update Using a Replication Controller. (2019). Recuperado de https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/

6.1. AUTO ESCALADOR: HPA

El auto escalador de *Kubernetes*, HPA por sus siglas en inglés (*Horizontal Pod Autoescaler*), es una funcionalidad nativa que escala de manera automática el número de *pods* en el clúster según unas reglas previas. Esta funcionalidad no es aplicable a objetos que no permitan el replicado como puede ser el motor de *Kubernetes* [40]. Aunque sí es posible crear más nodos trabajadores, y a efectos prácticos pueda parecer un replicado del motor, no es lo mismo ya que motor hay únicamente uno.

El HPA forma parte de la API de *Kubernetes* por lo que no es necesario una instalación específica de la herramienta. Su funcionamiento se basa en que un controlador del clúster ajusta periódicamente el número de réplicas para que se mantenga un valor medio predeterminado según los objetos que se controlan y las reglas aplicadas. Dicho valor puede ser controlado por *Kubernetes,* como el uso de CPU o memoria, o puede ser una métrica personalizada provista por un agente externo, ya sea un software o creada por el usuario.

La lógica de su implementación es similar a un bucle de control con un periodo de iteración definido por el controlador, que por defecto es de 15 segundos. Durante cada una de las iteraciones el controlador consulta la utilización del recurso solicitado y las compara según las reglas o métricas especificadas por el usuario. Para acceder a estas métricas el controlador hace uso de dos APIs, *Resource Metrics API* para las métricas basadas en utilización de recursos de los *pods* o *Custom Metrics API* para el resto de métricas. Gráficamente se puede observar el proceso en la Figura 46.

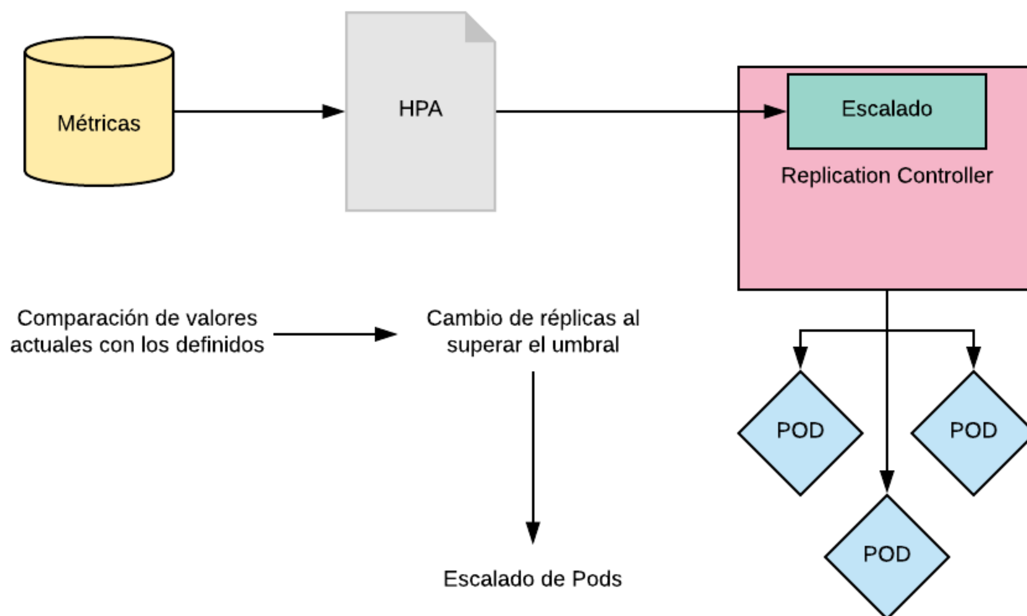

Figura 46: Funcionamiento del HPA

Para las métricas basadas en recursos, como puede ser CPU, el controlador lee las métricas devueltas por la API para cada uno de los *pods* en los que está funcionando el HPA. Con los datos obtenidos se realizan los cálculos necesarios para determinar el número de réplicas necesarias. Mientras que para las métricas personalizadas *Kubernetes* permite hacer uso de la API **autoescaling/v2beta2** para acceder a ellas.

[40] Horizontal Pod Autoscaler. (2019). Retrieved from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Los requisitos necesarios para la correcta lectura de las distintas métricas son:

- Tener activa la capa de agregación de la API, lo que hace que se puedan añadir las distintas APIs según el tipo de métrica.
- Añadir la API correspondiente a la capa mencionada anteriormente, que puede ser:
    - **metrics.k8s.io** para métricas basadas en recursos. Esta API es administrada por *metrics-server* que tiene soporte oficial de *Kubernetes.*
    - **custom.metrics.k8s.io** para métricas personalizas. La API es administrada por el adaptador del software utilizado.
    - **external.metrics.k8s.io** para métricas externas. De igual manera que la métrica anterior la API corre a cargo del adaptador del software externo elegido.
- Tener activo (*true*) el parámetro **--horizontal-pod-autoscaler-use-rest-clients** en la configuración del clúster. Por defecto las últimas versiones de *Kubernetes* lo tienen activado. De lo contrario se hace uso de *Heapster*, una funcionalidad obsoleta.

Para su uso, al igual que se hace con otros recursos del clúster, se utiliza la API de *Kubernetes*, **kubectl** como intermediaria entre el cliente y el motor. Los comandos básicos siguen la misma estructura que cualquier otro comando que haga uso de la API.

- Crear HPA
    - A partir de un archivo: **kubectl create -f <ARCHIVO>**
    - Sin archivo (solo posibilidad de escalado por CPU): Hace uso de autoescale anteriormente explicado
      `kubectl autoscale deployment <NOMBRE_HPA> --min= <PODS_MIN> --max=<PODS_MAX> --cpu-percent=<%_CPU>`
- Información del HPA
    - Básica: `kubectl get hpa <NOMBRE_HPA>`
    - Detallada: `kubectl describe hpa <NOMBRE_HPA>`
- Eliminar HPA
    - `kubectl delete hpa <NOMBRE_HPA>`

Si la creación es a través de un archivo, éste debe cumplir unas características básicas y asemejarse al mostrado a continuación:

```
apiVersion: autoscaling/v2beta/

kind: HorizontalPodAutoscaler

metadata:

  name: php-apache

  namespace: default

spec:

  scalteTargetRef:

    apiVersion: apps/v1

    kind: Deployment

    name: php-apache
```

```
minReplicas: 1

maxReplicas: 10

metrics:

-  type: Resource
   resource:
     name: cpu
     target:
       type: Utilization
       averageUtilization: 50
```

Como en todo archivo *.yaml* de *Kubernetes* la primera línea debe indicar la versión de la API que se utiliza. En este caso. como es el auto escalador, se hace uso de **autoscaling** y la versión **v2beta2**. El tipo (*kind*) se debe especificar como **HorizontalPodAutoescaler** para que el motor de *Kubernetes* sepa que tiene que crear un HPA. Los campos de *metadata* indican el nombre que tendrá el objeto y *namespace* en el que se ejecutará y los campos de la especificación (*spec*) deben indicar el objeto sobre el que se escala; en este caso un **deployment** de nombre **php-apache**, las réplicas tanto máximas como mínimas que puede tener el despliegue elegido y el tipo de recurso sobre el que se basa el escalado, que es **cpu**. Traduciendo el archivo, quiere decir que si se utiliza más del 50% de CPU disponible (**averageUtilization**) para el despliegue, el HPA actuará incrementando el número de réplicas.

6.2. USO DE MÉTRICAS PERSONALIZADAS

Como bien especifica la documentación de *Kubernetes* y se explica anteriormente, para el uso de métricas personalizadas es necesario hacer uso de un programa externo y el adaptador de dicho programa será el encargado de comunicarse con la API de *Kubernetes*. En este caso el programa utilizado es *Prometheus* que ha sido configurado en el capítulo anterior.

Con dicha configuración, *Prometheus* tiene la posibilidad de leer métricas personalizadas que trae por defecto, pero esas métricas no son accesibles por *Kubernetes*. El objetivo entonces es que el auto escalador del clúster acceda a la lectura hecha por *Prometheus* y según los datos leídos decida el escalado a realizar.

Para que el auto escalador de *Kubernetes* obtenga esos resultados y pueda decidir en qué momento escalar la aplicación es necesario realizar una configuración del clúster. El único requisito previo para ello es tener *Prometheus* desplegado y configurado correctamente para que recoja métricas de los objetos del clúster y accesible a través de una dirección IP. Es decir, la configuración de *Prometheus* que se explicó anteriormente.

Con este requisito cumplido se debe crear un adaptador que sirva como puente entre la API del clúster y *Prometheus*, el cual actúa de la siguiente manera[41]:

1. Descubre las métricas disponibles (*Discovery)*
2. Averigua con qué recursos de *Kubernetes* está asociada cada una de las métricas (*Association*)
3. Analiza cómo debe exponerla a la API de métricas personalizadas de *Kubernetes* para que pueda comprenderla (*Naming*)
4. Analiza cómo debe interactuar con *Prometheus* para obtener los valores reales (*Querying*)

[41] K8S Prometheus Adapter Configuration. (2019). Recuperado de https://github.com/DirectXMan12/k8s-prometheus-adapter/blob/master/docs/config-walkthrough.md

Para el correcto funcionamiento en el clúster, el flujo de trabajo consiste en:

- Crear un objeto **ApiService**, que a efectos prácticos es una extensión de la API de *Kubernetes*, para permitir que ésta pueda leer más valores que los predeterminados
- Asignarle los permisos necesarios a dicha API mediante roles para que pueda comportarse como un administrador y así acceder a todos los objetos del clúster. En la configuración se especifica el campo `resources: ["*"]` y `verbs: ["*"]`, lo que le da permiso para ejecutar todo tipo de acciones sobre cualquier recurso.
- Crear un despliegue que contenga la imagen de *Prometheus* así como la configuración necesaria como la URL donde se expone el operador. Opcionalmente se puede crear un **ConfigMap** para enlazar un archivo de configuración.
- Crear un servicio para exponer el despliegue creado anteriormente
- Asignar rol al servicio anterior para darle los permisos necesarios.

Para la instalación en un clúster de *Kubernetes* hay que seguir los pasos descritos por los desarrolladores, disponible en:

https://github.com/DirectXMan12/k8s-prometheus-adapter/tree/master/deploy

Donde se especifica que hay que crear una imagen a partir de un *Dockerfile*, generar un **secret** para la autentificación del tráfico HTTPS y aplicar todos los archivos ***.yaml*** que incluye el repositorio mediante `kubectl apply -f < ... >`.

Estos archivos requieren pequeñas configuraciones ya que no está automatizado el proceso. En la configuración del despliegue hay que especificar la URL del operador de *Prometheus* y el archivo **ConfigMap** si se desea.

Una vez realizado este proceso se crea la API `custom.metrics.k8s.io` a la que se puede acceder y comprobar las métricas de las que dispone como se muestra en la Figura 47, donde únicamente se muestran unas pocas de todas las resultantes.

```
root@ubuntu:/# kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1
{"kind":"APIResourceList","apiVersion":"v1","groupVersion":"custom.metrics.k8s.io/v1beta1","resources":[{"name":"pods/cpu_loa
d_average_10s","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"pods/last_seen","singul
arName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"pods/memory_usage_bytes","singularName":"","n
amespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"namespaces/memory_swap","singularName":"","namespaced":fal
se,"kind":"MetricValueList","verbs":["get"]},{"name":"pods/spec_memory_limit_bytes","singularName":"","namespaced":true,"kind
":"MetricValueList","verbs":["get"]},{"name":"namespaces/spec_memory_swap_limit_bytes","singularName":"","namespaced":false,"
kind":"MetricValueList","verbs":["get"]},{"name":"namespaces/cpu_system","singularName":"","namespaced":false,"kind":"MetricV
alueList","verbs":["get"]},{"name":"namespaces/fs_sector_reads","singularName":"","namespaced":false,"kind":"MetricValueList"
```

Figura 47: Lista de métricas personalizadas accesibles por la API de Kubernetes

En cuanto se añada un objeto sobre el cual se puedan leer métricas, *Prometheus* lo descubrirá y podrá interpretarse mediante la API y, por tanto, el auto escalador podrá trabajar en función de los resultados leídos.

6.3. PRUEBAS DE CARGA CON HEY

Todo cambio en el desarrollo de cualquier herramienta software conlleva unas pruebas que permitan verificar el correcto funcionamiento de dicho cambio. Para comprobar el correcto funcionamiento del auto escalador es necesario hacer pruebas de carga según el recurso que se monitoriza.

Una opción para realizar pruebas de carga puede ser lanzar peticiones al servicio de *Kubernetes* en un bucle infinito desde la consola como se muestra en la Figura 48.

```
root@ubuntu:~# while true; do curl http://192.168.99.100:32268; done
```

Figura 48: Prueba de carga a un servicio mediante la línea de comandos

De esta forma se obtiene el rendimiento del servicio en situaciones de estrés ya que se le aplica una carga constante de peticiones (Figura 49).

```
root@ubuntu:~# while true; do curl http://192.168.99.100:32268; done
Hello World!Hello World!Hello World!Hello World!Hello World!Hello World!Hello W
orld!Hello World!Hello World!Hello World!Hello World!Hello World!Hello World!He
llo World!Hello World!Hello World!Hello World!Hello World!Hello World!Hello Wor
ld!Hello World!Hello World!Hello World!Hello World!Hello World!Hello World!Hell
o World!Hello World!Hello World!Hello World!Hello World!Hello World!Hello World
!Hello World!Hello World!Hello World!Hello World!Hello World!Hello World!Hello
World!Hello World!Hello World!Hello World!Hello World!Hello World!Hello World!H
ello World!Hello World!Hello World!Hello World!Hello World!Hello World!Hello Wo
```

Figura 49: Respuesta ante las peticiones HTTP

Pero en ocasiones un único escenario no es suficiente para conocer el comportamiento real, ya que, por ejemplo, podría darse la posibilidad que ante peticiones constantes el auto escalador funcione y por lo tanto creamos que está bien configurado, pero ante peticiones masivas en intervalos espaciados de tiempo no reaccione, lo que colapsaría la aplicación. Una solución a ello es utilizar programas que realicen las cargas pudiendo especificar opciones de comportamiento. La herramienta más utilizada por la comunidad de *Kubernetes* para hacer pruebas sobre los servicios del clúster es un software llamado ***Hey***.

Hey es un pequeño programa que lanza carga de trabajo a una aplicación web[42].

Únicamente es necesario un comando para instalarlo:

```
go get -u github.com/rakyll/hey
```

Y su uso por gran parte de la comunidad es gracias a las opciones que permite en su ejecución según los parámetros introducidos, siendo los más interesantes:

- -n: Número de peticiones a enviar.
- -c: Número de peticiones concurrentes a enviar.
- -z: Duración máxima del tiempo de envío. Si se supera el tiempo máximo se cancela la ejecución.
- -m: Método HTTP entre GET, POST, PUT, DELETE, HEAD y OPTIONS.
- -t: Tiempo de espera para la respuesta de las peticiones.
- -a: Autenticación básica de la forma usuario:contraseña.
- -x: Uso de proxy.
- -cpus: Número de CPUs a utilizar.

Como ejemplo se han lanzado 1000 peticiones. La salida muestra, no solo los datos que aparecen en la Figura 50: los tiempos referentes al envío y un histograma de respuesta. Sino también disponde de más opciones como la distribución de la latencia, código del estado (200 si es correcto) y otros detalles.

[42] Hey. (2019). Recuperado de https://github.com/rakyll/hey

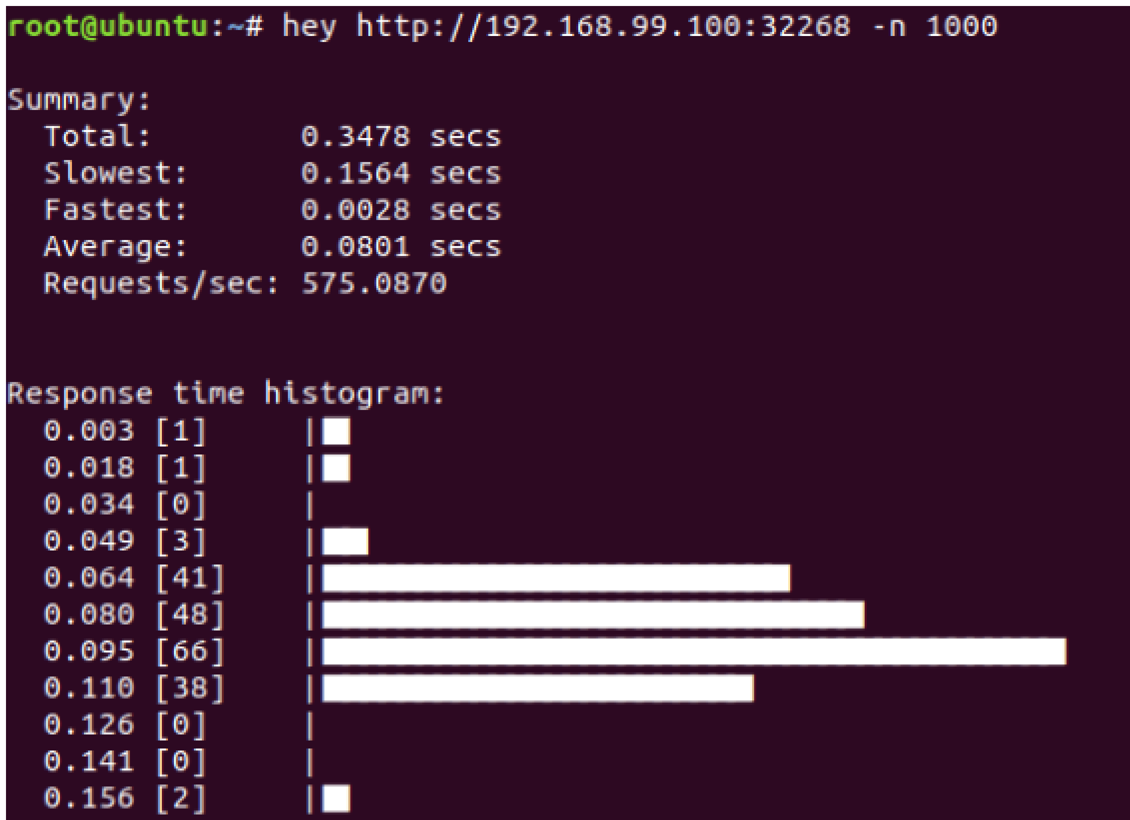

Figura 50: Respuesta de Hey tras realizar un proceso

El uso de esta herramienta permite simular distintos escenarios para poner a prueba el comportamiento tanto de las aplicaciones sobre las que se lanzan las peticiones como de los objetos relacionados con el HPA.

CAPÍTULO 7

ESPECIFICACIÓN, DISEÑO Y DESARROLLO DE UNA INTERFAZ WEB

7. ESPECIFICACIÓN, DISEÑO Y DESARROLLO DE UNA INTERFAZ WEB

Como se propuso inicialmente, se quiere realizar una aplicación web que facilite el envío de peticiones HTTP mientras se observa el comportamiento de varios objetos del clúster. La aplicación no debe intentar ser un programa de monitorización o similar, sino únicamente una interfaz que sirva como referencia para conocer el comportamiento del clúster frente a distintas cargas de trabajo. Ya que existen distintas herramientas en el mercado, incluidas las propias de *Kubernetes,* no se intenta competir con ellas ya que la principal novedad que propone esta aplicación es compartir en una misma ventana, sin necesidad de tener abiertos distintos terminales o programas, la visualización simultánea de dos objetos del clúster (despliegue y HPA), junto con el envío de peticiones para posteriormente observar los resultados obtenidos.

Para ello se han seguido distintas etapas de la ingeniería del software, empezando por el análisis de la aplicación, qué se quiere realizar y cómo, pasando por diseño, codificación y por último las pruebas unitarias para comprobar el correcto funcionamiento.

7.1. ANÁLISIS

7.1.1. ESPECIFICACIONES FUNCIONALES

La aplicación debe permitir al usuario elegir la configuración de las peticiones a realizar y los objetos del clúster implicados: el despliegue sobre el que realizar el envío de las peticiones, el *namespace* al que pertenece, el servicio sobre el cual acceder y el escalador que lo controla. Para ello se enumeran las especificaciones funcionales implementadas en el sistema en la Tabla 3.

#	Nombre	Descripción
1	Indicar peticiones totales	El sistema configurará el envío de peticiones según la totalidad de ellas
2	Indicar peticiones concurrentes	El sistema configurará el envío de peticiones según cuantas de ellas se envían concurrentemente
3	Indicar peticiones por segundo	El sistema configurará el envío de peticiones según cuantas de ellas se envían cada segundo como máximo
4	Elegir servicio	El sistema configurará el envío de peticiones para realizarlo sobre la URL indicada
5	Elegir *namespace*	El sistema mostrará los objetos del clúster según el *namespace* seleccionado
6	Elegir despliegue	El sistema mostrará los datos de replicado del despliegue seleccionado
7	Elegir HPA	El sistema mostrará los datos de replicado del auto escalador seleccionado
8	Enviar petición	El sistema iniciará el proceso de envío de peticiones con la configuración indicada por el usuario
9	Borrar datos	El sistema borrará los datos mostrados hasta el momento

Tabla 3: Especificaciones funcionales de la aplicación

7.2. DISEÑO

7.2.1. DIAGRAMA DE CASOS DE USO

Las distintas funcionalidades mencionadas anteriormente que puede realizar un usuario vienen recogidas en el siguiente diagrama de casos de uso (Figura 51).

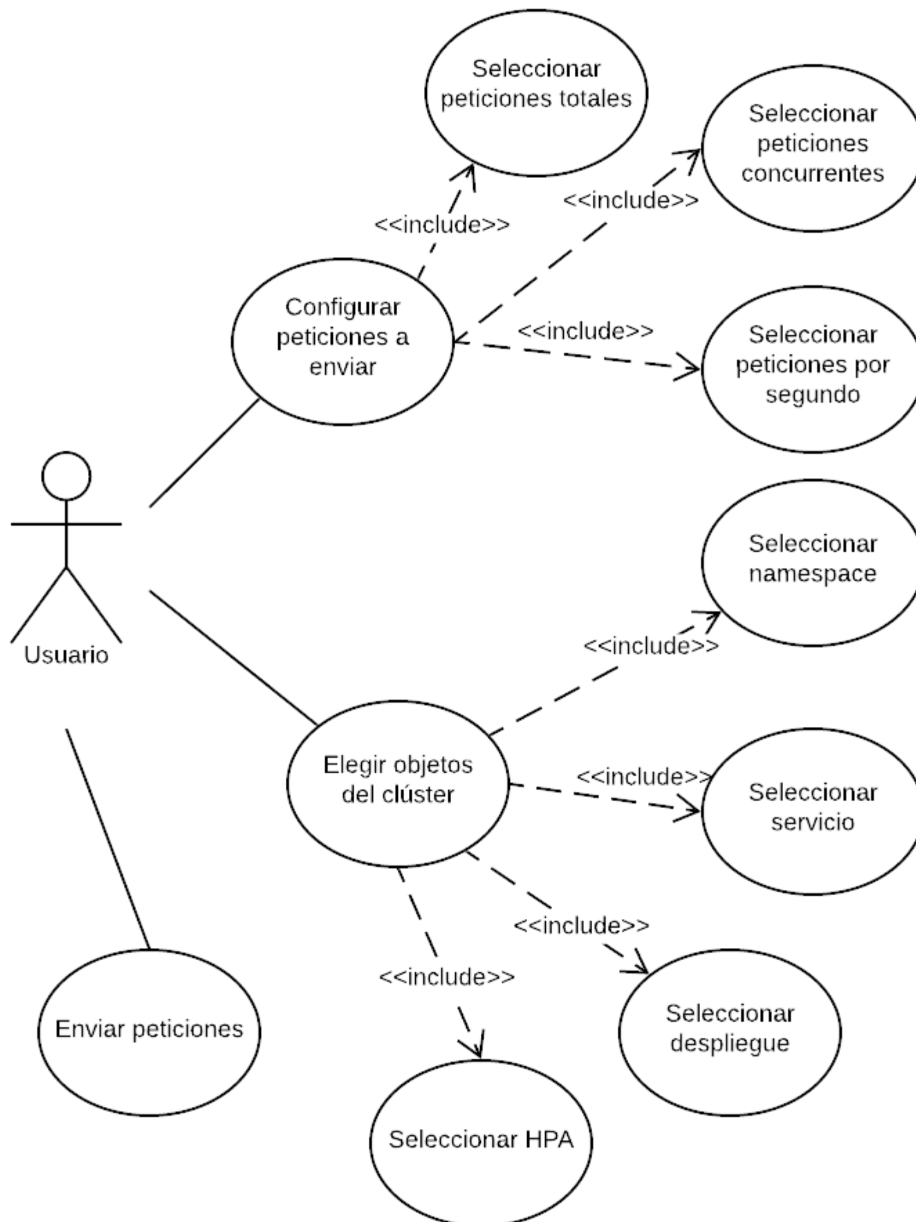

Figura 51: Diagrama de Casos de Uso

7.2.2. PROTOTIPOS DE INTERFAZ

Conocida la funcionalidad, se crean distintos prototipos de interfaz que cumplan con distintas normas de usabilidad, así como los requisitos de la aplicación. Para ello se crearon diversas opciones de las dos pantallas con las que cuenta la aplicación, teniendo la pantalla donde se muestran los resultados una pequeña variación según el estado en que se encuentre, ya que no mostrará lo mismo cuando esté en proceso de envío de peticiones que cuando ya termine, puesto que mostrará los resultados correspondientes.

Para ello se generaron distintos *mockups*, siendo elegidos los que se muestran a continuación (Figura 52, Figura 53 y Figura 54).

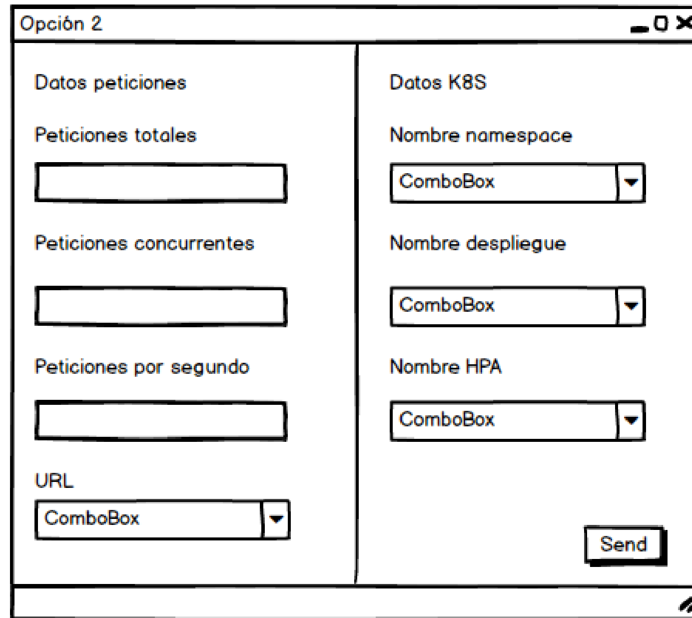

Figura 52: Mockup interfaz principal

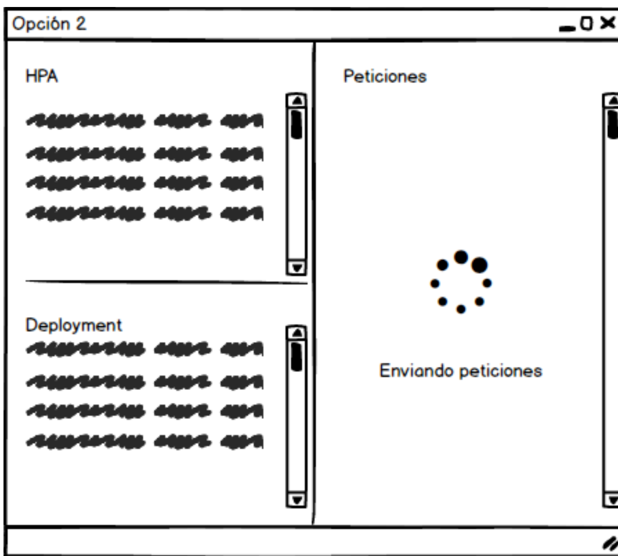

Figura 53: Mockup interfaz enviando peticiones

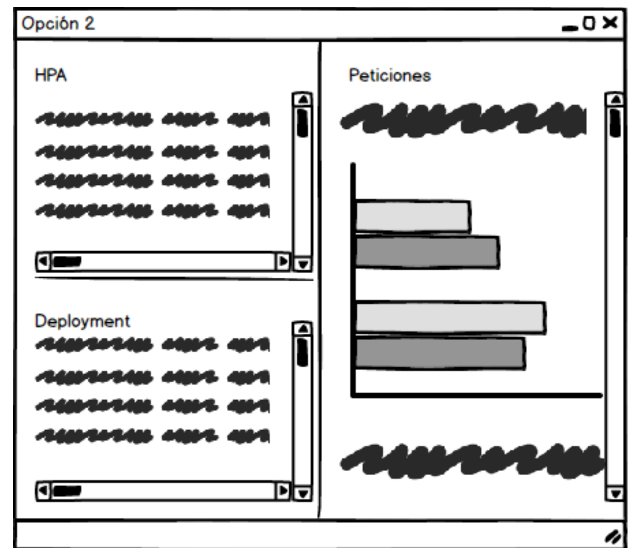

Figura 54: Mockup interfaz mostrando datos

7.3. IMPLEMENTACIÓN

Tras la definición de la aplicación que se quiere construir tanto a nivel funcional como visual se debe realizar la implementación de la misma. Para poder realizarla de manera correcta son necesarios unos recursos mínimos que dependen más del clúster del que se leen datos que de la aplicación desplegada en sí. Estos requisitos son:

- **Cliente con navegador web** (o accesible desde el exterior) ya que la aplicación requiere de interacción directa, pudiendo elegir entre:
    - Mozilla Firefox
    - Google Chrome
    - Internet Explorer
    - Opera
    - Safari
- **Recursos *hardware***: Al menos 4GB de RAM para la instalación de *Docker* y 2 *cores* para levantar el nodo maestro del clúster además de ser recomendable una arquitectura de 64 bits. La RAM necesaria así como el espacio en disco dependen de las aplicaciones a levantar en el clúster. Cuanto más pesadas sean mayor espacio se necesitará, pero una configuración inicial con los recursos anteriores funciona correctamente.
- **Sistema operativo** donde ejecutar *Docker*:
    - Windows 10
    - Mac OSX
    - Sistemas derivados de UNIX
- **Servidor web**:
    - Apache (recomendado)

Dentro de estos requisitos se puede elegir cualquier combinación posible. Para la realización de este proyecto se ha decidido utilizar un sistema Linux Ubuntu de 8GB de RAM y 150 GB de disco duro. La aplicación se levanta en un servidor Apache y es accesible a través de **localhost**.

7.3.1. INICIAR LA APLICACIÓN EN UN SERVIDOR

Cumpliendo todos los requisitos anteriores se puede iniciar la aplicación y observar la interfaz. Para ello es necesario mover el archivo **.war** generado en la carpeta de *Tomcat*.

Para generar el archivo **.war** haremos uso de Maven ya que al ser necesario un único comando facilita el proceso. En el directorio donde se encuentre el archivo **pom.xml** ejecutamos el comando

```
$ mvn clean package
```

Y la salida, si la construcción ha sido correcta, debe mostrar **BUILD SUCCESS** junto con más detalles, como el tiempo total de construcción o la hora final tal y como se muestra en la Figura 55.

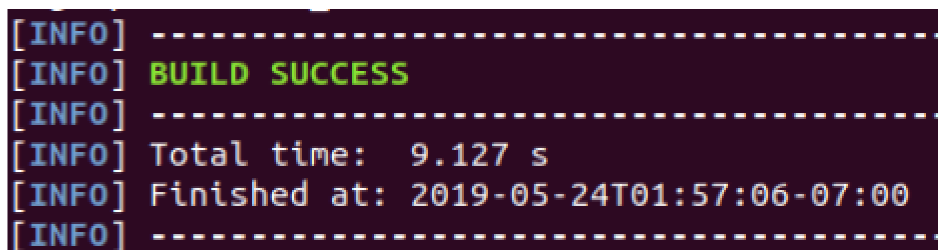

Figura 55: Construcción correcta del proyecto

Una vez empaquetada la aplicación es necesaria introducirla en el servidor web. Para ello primero introduciremos el archivo recién construido mediante el comando `mv` (o `cp` para copiar) con destino la carpeta **webapps** de *Tomcat* para después poner en marcha el servidor (suponiendo que no lo esté ya) ejecutando un archivo **.sh** dentro de la carpeta **bin** del servidor.

Para facilitar el proceso se ha decidido crear un script *bash* que automatice el proceso de construcción y copiado a la carpeta destino tal y como se muestra a continuación:

```
#!/bin/bash

#Ir al directorio donde está el proyecto

cd /home/ubuntu/TFGDocker/GUI/K8S_Interface/Interface_K8S

#Correr con maven para generar el .war

mvn clean package

#Mover el .war a la carpeta de tomcat

mv
/home/ubuntu/TFGDocker/GUI/K8S_Interface/Interface_K8S/target/Interf
ace_K8S-0.0.1.war /opt/tomcat/apache-tomcat-9.0.16/webapps

#Iniciar el servidor

sh /opt/tomcat/apache-tomcat-9.0.16/bin/startup.sh
```

Una vez que la aplicación está en el servidor, se puede acceder a ella mediante la ruta configurada en *Tomcat* (por defecto el puerto 8080) mostrándose la interfaz de la Figura 56.

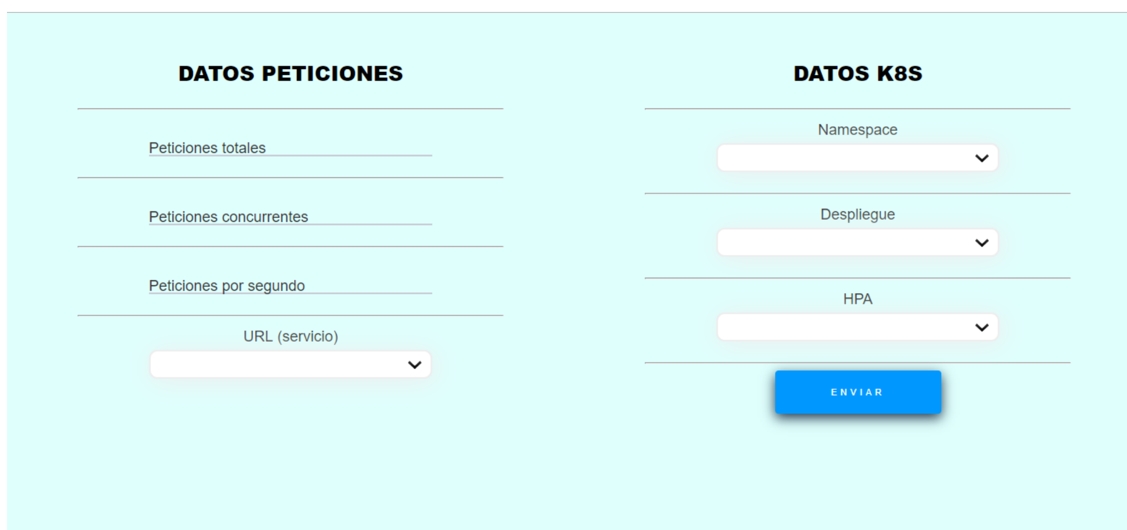

Figura 56: Interfaz web principal

Al no haber ningún objeto en ejecución en el clúster, los campos “URL”, “*Namespace”*, “Despliegue” y “HPA” están vacíos y no contienen información. Si se pulsa el botón “Enviar” se navega a la siguiente pantalla donde se muestran los datos de los objetos elegidos mientras se realizan las peticiones masivas por HTTP, siendo la interfaz la mostrada en la Figura 57.

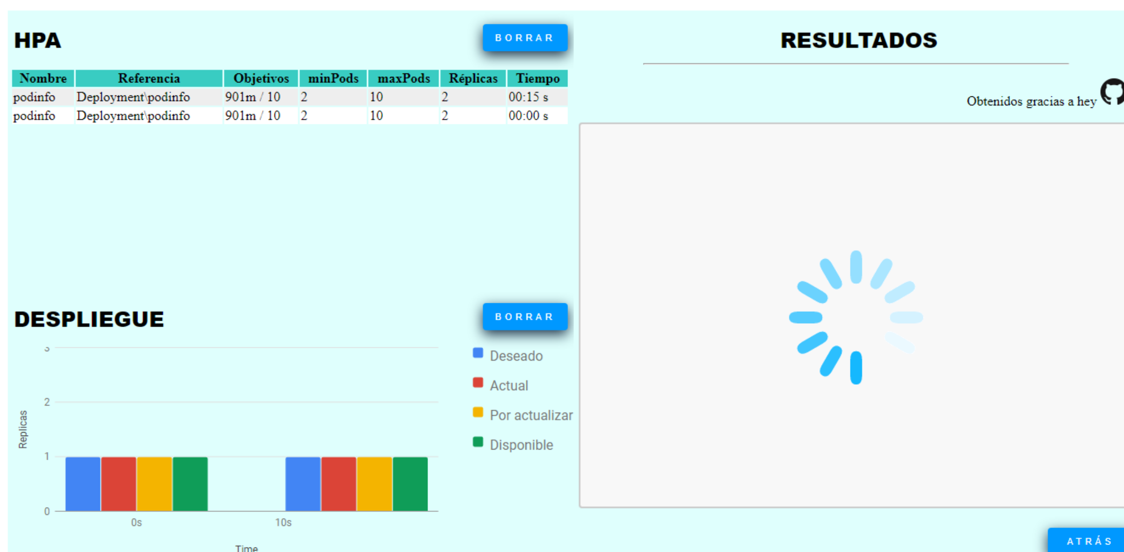

Figura 57: Interfaz web mandando peticiones

En la parte izquierda se muestran los datos referentes al despliegue y el HPA seleccionado mientras que a la derecha se muestran los resultados obtenidos por el software **Hey** tras las peticiones HTTP.

7.3.2. DETALLES DE IMPLEMENTACIÓN

Se ha utilizado lenguaje Java para el *backend* mientras que el *frontend* está escrito con lenguajes web: JSP que incluye HTML y CSS para el estilo. Como IDE se ha hecho uso de Eclipse.

La aplicación, como se explica anteriormente, tiene la finalidad de leer datos del clúster de *Kubernetes* creado anteriormente y mostrarlos por pantalla para que el usuario los disponga con mayor facilidad, todo ello a la vez que se envían peticiones HTTP por lo que en una misma pantalla se dispone del comportamiento de los distintos objetos tras la carga de trabajo así como los datos de las peticiones enviadas.

La lógica de funcionamiento de la aplicación se basa en aprovechar la salida en formato JSON de los objetos del clúster para leer los datos necesarios y poder mostrarlos. La lectura de los datos se guardará en clases creadas con los atributos necesarios.

Cuando el usuario inicia la aplicación, ésta carga internamente los datos necesarios a mostrar, primero todos los *namespaces* y seguidamente los objetos (servicios, despliegues y HPA) que pertenezcan al espacio de nombres que se haya elegido, que por defecto es *default*. Si se elige otro *namespace*, se recargan el resto de objetos para mostrar únicamente los que pertenezcan al elegido. Esta carga de datos se hace llamando a una clase llamada **ReadJSON.java** que ejecuta el comando necesario, almacena la respuesta recibida en formato JSON y devuelve la información necesaria, en este caso únicamente los nombres que mostrar.

Con los datos cargados y mostrados en la pantalla el usuario puede elegir la configuración de la carga de trabajo a enviar, esto es, número de peticiones totales, concurrentes y por segundo. Si no se introducen valores el programa al que se llama para realizar las peticiones añade unos por defecto, 200 peticiones totales, 50 concurrentes sin límite de peticiones máximas por segundo. Además, se ha codificado el *frontend* de tal manera que se le impida al usuario ingresar valores fuera de rango como podrían ser letras. Así se evita la comprobación en el *backend* ya que nos aseguramos que los valores que llegan son enteros.

Una vez que el usuario introduce todos los valores que considere necesarios, al pulsar el botón *“ENVIAR”* el programa crea un objeto **Peticion.java** con atributos que corresponden a todos los datos introducidos y se llama al constructor de la clase **Results.java**, clase que ejecuta tres hilos para realizar operaciones concurrentes, siendo éstas:

1. Hilo para lanzar peticiones: Según los datos recibidos en el objeto **Peticion** se genera el comando a ejecutar y crear la carga de trabajo correspondiente. Si no se hiciese este trabajo en un hilo concurrente bloquearía la aplicación principal y por tanto no se podría observar toda la información de manera simultánea.
2. Hilo para leer datos del despliegue: Este hilo llama a un método de la clase **ReadJSON.java** para que, según el despliegue y *namespace* indicados, ejecute el comando correspondiente y obtener la respuesta en formato JSON. Esta respuesta permite crear un objeto **DeploymentK8S.java** con los atributos obtenidos, que lo almacena en un *ArrayList* y es leído por el *frontend*, conteniendo este *ArrayList* la información que se muestra en la tabla.
3. Hilo para leer datos del HPA: De igual manera que el hilo anterior se llama a un método de la clase **ReadJSON.java** que crea un objeto **HpaK8S.java** con sus propios atributos. Este objeto se añade, de igual manera que el hilo anterior, a un *ArrayList* del cual se leerán los datos para mostrarse en el *frontend*.

La ejecución durará hasta que el usuario cierre la aplicación o pulse el botón “*ATRÁS”*. De manera excepcional también pueden borrarse los datos mostrados hasta el momento pulsando el botón *“BORRAR”,* lo que hará que se vacíe el *ArrayList* correspondiente y por tanto la información a mostrar. Dicha información se ha implementado de manera que se actualiza cada 15 segundos, la mitad de tiempo que el intervalo de actualización que tiene *Kubernetes* por defecto.

En la Figura 58 se muestra un diagrama de secuencias como idea general del funcionamiento ya que no se especifican los métodos exactos sino únicamente la lógica que debe seguir la implementación. Para indicar el paralelismo en el código se utiliza la etiqueta *par*.

Adicionalmente, como herramienta de control de versiones se ha hecho uso de *Git*.

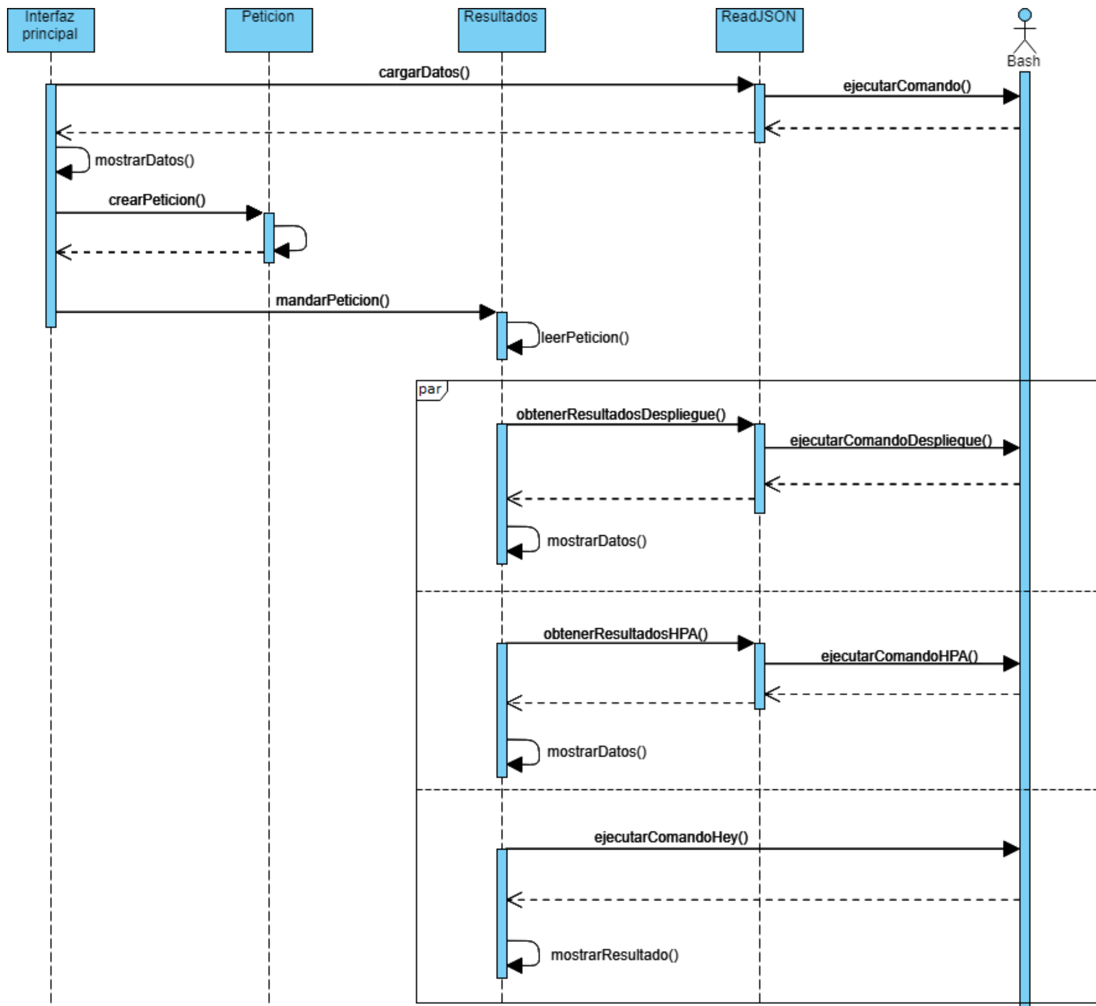

Figura 58: Diagrama se secuencias

7.3.2.1. DESARROLLO INTERFAZ PRINCIPAL

Para la primera pantalla que se encuentra el usuario al iniciar la aplicación se deben hacer uso de distintas clases tanto **.java** como **.jsp.** Para el funcionamiento se debe tener en cuenta que en el momento en el que el usuario decida elegir un *namespace* distinto al que se muestra por defecto se deben cargas los datos asociados al nuevo *namespace* elegido. Esta carga de datos supone un problema ya que se necesita de un modelo asíncrono en un entorno que no lo es. El modelo de programación en JSP por defecto es síncrono, es decir, la visualización que obtienen los usuarios (lado cliente) es el estado del lado servidor en un momento concreto. Al interaccionar con la interfaz y necesitar de nuevos datos que no están cargados en el *frontend* es necesario una llamada asíncrona al servidor que actualice los datos de la interfaz.

Existen distintas maneras de superar este problema en concreto, incluso sin hacer uso de asincronía, ya que se podrían cargar todos los datos de todos los *namespaces* y guardarlos en un mapa clave-valor para tenerlos accesibles desde el lado cliente y así al elegir uno distinto no es necesario cargar datos desde el servidor. Pero para tener un código más limpio y escalable a futuros cambios se ha decidido utilizar funciones asíncronas haciendo uso de AJAX (**A**synchronous **J**avaScript **A**nd **X**ML).

AJAX permite actualizar el lado servidor sin necesidad de recargar la interfaz entera, lo que posibilita actualizar ciertas funcionalidades del cliente sin necesidad de recargar la ventana por completo.

El flujo de trabajo de AJAX, como se muestra en la Figura 59 comienza cuando ocurre un cambio en el lado del cliente (un botón pulsado, un evento, etc.), entonces es cuando JavaScript genera un objeto XMLHttpRequest con información que se envía al servidor. El servidor analiza este objeto, realiza las acciones necesarias y responde de vuelta al lado cliente, siendo esta respuesta leída por JavaScript y realizando las acciones correspondientes.

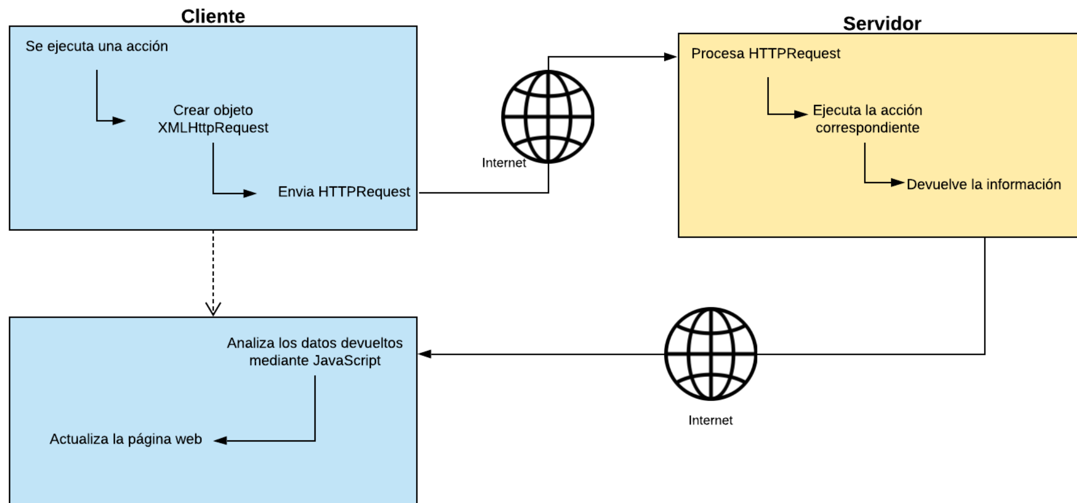

Figura 59: Funcionamiento de AJAX

Por lo que, siguiendo esta lógica, en la aplicación se tienen tres campos que deben cambiar asíncronamente cada vez que se seleccione un nuevo *namespace*: Despliegues, servicios y HPA.

Para llevarlo a cabo se crea un evento en el objeto ComboBox para actuar en el momento que se cambie su valor

```
<select name="namespace" onChange="refreshComboBox()">
```

Este evento llama a una función JavaScript que a su vez llama a las funciones correspondientes a cada objeto que hay que actualizar.

```
function refreshComboBox() {
        refreshComboBoxDeployment()
        refreshComboBoxHPA()
        refreshComboBoxService()
}
```

Cogiendo como ejemplo la primera función, ya que todas siguen la misma lógica cargando cada una de ellas datos distintos del servidor, dentro del método, siguiendo el flujo de datos indicado anteriormente, lo primero es crear un objeto XMLHttpRequest.

```
var requestRefreshComboBox = new XMLHttpRequest();
```

Con este objeto creado se envía información al servidor, en este caso se hace uso de una página JSP que permite ejecutar código Java pasando por parámetro los datos necesarios.

```
var urlRefreshComboBox =
"RefreshComboBoxDeployment.jsp?namespace="+n;
requestRefreshComboBox.open("GET", urlRefreshComboBox, true);
requestRefreshComboBox.send();
```

Enviada la información, en este caso el nuevo *namespace* elegido, el servidor ejecuta las acciones necesarias.

```
<%
  //Leer atributo recibido
  String[] str = request.getParameterValues("namespace");
  String namespace = str[0];
  //Realizar acciones necesarias
  //Devolver nuevo elemento HTML
  out.write(strDeployment);
%>
```

Cuando el servidor termina de realizar su acción es turno de JavaScript de leer los nuevos datos en el lado cliente. Según se ha realizado en la aplicación, JavaScript escribe código HTML sobre el objeto indicado mostrando así los nuevos datos.

```
if (requestRefreshComboBox.readyState == 4) {
  //Leer respuesta recibida
  var ret = requestRefreshComboBox.responseText;
  //Modificar elemento correspondiente con
  //el texto devuelto por el servidor
  document.getElementById('comboBoxDeployment').innerHTML = ret;

}
```

Esta acción se realiza para los distintos objetos que se ven afectados por el cambio de datos en el *frontend*, consiguiendo así una concordancia entre los valores mostrados al usuario y los reales que pertenecen al clúster que se está ejecutando en segundo plano.

7.3.2.2. DESARROLLO INTERFAZ DE RESULTADOS

En cuanto a la pantalla que muestra los resultados existen distintos detalles a tener en cuenta. Como se explica anteriormente al acceder a esta parte del programa se lanzan hilos para realizar tareas concurrentes ya que de lo contrario no es posible mostrar los resultados de manera simultánea.

Un hilo obligatorio es el que manda las peticiones HTTP ya que mientras realiza el proceso no puede computar otras operaciones. Para la creación del hilo en Java se hace uso de la clase `Thread` de la forma:

```
public void lanzarHilo(Peticion p) {
  Thread t = new Thread() {
    public void run() {
      metodoConcurrente(p);
    }
  };
}
```

Al disponer del objeto *Peticion*, que tiene como atributos los distintos datos que el usuario ha introducido en la ventana principal, únicamente es necesario ejecutar un método que realice las operaciones necesarias. Ese método será diferente según la tarea que se quiera realizar, para el envío de peticiones HTTP únicamente se ejecutará el comando necesario mientras que en el caso de tener que mostrar el estado de los distintos objetos del clúster estará formado por un bucle `while` infinito que realiza consultas al estado del clúster para devolver los datos necesarios además de un tiempo de espera definido en 15 segundos mediante la función `Thread.sleep(15000)`. Una vez definido el hilo con el método que tiene que ejecutar se inicializa llamando mediante la función `thread.start();`

Una vez funcionando la lectura de datos de manera concurrente en el *backend*, es necesario mostrarlos al usuario en el *frontend*, para ello se realizan mediante JavaScript llamadas asíncronas cada cierto tiempo en las que se devuelven los datos leídos. El lado servidor de la aplicación genera un `ArrayList<ObjetoK8S>` donde `ObjetoK8S` es una clase creada y cada posición del `ArrayList` corresponde con los datos a mostrar en una fila de la tabla, leyendo en cada iteración la información correspondiente. De esta manera se actualiza la tabla mediante una inyección de código Java en una clase JSP de la siguiente manera:

```
<% for(int i = array.size()-1; i >= 0 ; i--){%>
<tr>

  <td><%=array.get(i).getAtributo1()%></td>
  <td><%=array.get(i).getAtributo2()%></td>
  <td><%=array.get(i).getAtributo3()%></td>
</tr>

<%}%>
```

Consiguiendo crear una nueva fila de la tabla por cada elemento existente en la colección. Además, se recorre la lista de forma inversa para que se muestre en primer lugar siempre el último dato leído.

Siguiendo esta lógica de programación se consigue mostrar los datos necesarios de uno de los dos hilos mencionados anteriormente teniendo en cuenta que las llamadas al servidor son de manera asíncrona por lo que hay que incluir los pasos mencionados anteriormente a esta solución. Para el otro hilo que lee datos del clúster la idea es similar, pero al mostrarse gráficos en lugar de una tabla cambia levemente la forma de implementación.

La manera de mostrar los datos del auto escalador en la interfaz se ha decidido que sea mediante gráficos de barras para obtener un resultado con más impacto visual. Estos gráficos han sido realizados en JavaScript haciendo uso de la librería "*Google Charts*"[43]. La lógica de implementación es igual que la anterior, de manera asíncrona se leen los datos que el hilo recoge del clúster y en una clase JSP se inyecta el código Java que permite traducir esos datos a HTML. La diferencia reside en la manera en que se traduce la lista con los valores a gráfico de barras ya que éste lee una entrada con la estructura "[0, … , 0]" donde cada valor corresponde a una barra. Es por ello que el código Java inyectado debe ser de la forma que se muestra a continuación.

---

[43] Charts | Google Developers. (2019). Recuperado de https://developers.google.com/chart/

```
<%
String str ="";
for(int i=0; i<array.size(); i++){
  ObjetoK8S o = array.get(i);
  str+="[" + o.getAtributo1() + ", "  + ...
  ... + ", " + o.getAtributoN()+"]";
}
out.write(str);
%>
```

Así se consigue devolver los atributos leídos del objeto del clúster de manera que la API de Google Charts pueda transformarlo en un diagrama de barras. Solo quedaría mostrar en el lado del cliente el mensaje recibido. Para ello se hace uso de la función `eval` de JavaScript que permite transformar la cadena de texto entrante en un array de enteros de la forma `eval("[" + entryData + "]")`, siendo `entryData` la respuesta generada con el código anterior.

Mientras que el tercer y último hilo, el que ejecuta las peticiones HTTP, se compone únicamente de una llamada asíncrona que se mantiene en espera de los resultados mientras una variable no sea **true**. Esta variable se activará cuando el servidor tenga la respuesta de la línea de comandos y hará que la llamada asíncrona pueda devolver un mensaje (el resultado) que mostrar por pantalla.

Con la implementación de los procesos que ejecutan los hilos finaliza la implementación de la funcionalidad del programa, en este punto el programa permite al usuario realizar todas las acciones que se propusieron. A falta del cambio de estilo visual de la aplicación queda comprobar que los resultados que obtiene la aplicación se corresponden con los reales, es decir, que la aplicación funciona bien mediante pruebas.

7.4. PRUEBAS

Una vez terminada la codificación de la aplicación es necesario realizar *tests* para comprobar que el funcionamiento es el adecuado. Ya que los valores con los que trata la aplicación casi en su totalidad son efímeros, no se puede comprobar de manera automática que funciona correctamente, sino que la solución para estos datos es comprobar la salida de la aplicación con la salida de la API de *Kubernetes*.

Los datos que no se pueden comprobar de manera automática son los que definen el comportamiento de los objetos en un momento dado, es decir, el tiempo que lleva el despliegue en ejecución, los recursos que está consumiendo en un preciso momento, etc. Sin embargo sí hay ciertos valores que no cambian en el tiempo y se puede comprobar si son correctos, como la lectura de los objetos existentes ya que de un momento a otro no es normal (a menos que se esté realizando otro proceso) que se cree o se destruya por ejemplo un *namespace* o un servicio, lo que hace posible comprobar si la lectura que hace la aplicación se corresponde con los valores devueltos por la API.

La lógica de implementación para los *tests* es ejecutar un comando que devuelve únicamente los parámetros indicados de un objeto en concreto y comprobarlos con la lectura que hace la aplicación. Por ejemplo, para comprobar la correcta lectura de los despliegues existentes se le pregunta a la API por el nombre de los mismos y se comprueba que los despliegues leídos con la aplicación coinciden en número total y nombre de cada uno de ellos.

El comando introducido para obtener todos los objetos es el siguiente:

```
kubectl get OBJETO -o=jsonpath='{.items[*]}{ATRIBUTO}'
```

Introduciendo las variaciones necesarias como por ejemplo añadir {" "} al final para que separe el texto devuelto por espacios, se pueden obtener los atributos especificados y así poder comprobar que se corresponden, teniendo en cuenta que la salida contendrá un espacio al final que el test lo leerá como objeto, por lo que hay que obviarlo.

Siendo cada test resultante similar al mostrado a continuación:

```
@Test
public void testReadDeployments(){
  //Leer datos de la API
  String[] salida = ejecutarComando(...).split(" ");
  //Eliminar último caracter
  String[] strAux = new String[salida.length-1]
  System.arraycopy(salida,0,strAux,0,salida.length-1);
  //Comprobar que se corresponde con la aplicación
  boolean test = true;
  //Si no tienen el mismo tamaño = false
  if(arrAux.length != metodo_a_probar(...).length) test =
false
  //Si no tienen los mismos objetos = false
  for(String s : strAux){
      if(!array.contains(s)){
          test = false;
          break;
      }
  }
  assertEquals(test,true);
}
```

Donde según el método a comprobar, el test realiza la configuración necesaria y así abarcar las distintas opciones posibles.

El número total de *tests* escritos son cuatro, los correspondientes con la lectura de objetos del clúster: servicios, *namepsaces*, despliegues y HPA. Así se comprueba que los datos mostrados al usuario corresponden realmente con los datos que se encuentran en el clúster estando distribuidos de la siguiente forma:

Nombre test		Método al que se aplica
`testReadNamespace()`	→	`ArrayList<NamespaceK8S> readNamespace()`
`testReadDeployments()`	→	`ArrayList<DeploymentK8S> readDeployments(String n)`
`testReadHPA()`	→	`ArrayList<HpaK8S> readHPA(String n)`
`testReadServices()`	→	`ArrayList<ServiceK8S> readServices(String n)`

De esta manera, ejecutando el comando `mvn test` se ejecutan los tests y si se construyen correctamente se muestra por pantalla la salida de la Figura 60 donde se puede ver que los 4 tests de la clase **ReadJSONTest** han sido correctos. De esta manera podemos asegurar que la aplicación lee los datos del clúster de igual manera que lo hace la API de *Kubernetes*.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running test.ReadJSONTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 133.297 sec

Results :

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  02:46 min
```

Figura 60:Tests pasados con éxito

En cuanto a las pruebas de la interfaz (no de funcionalidad) no se han considerado necesarias ya que, como se explica anteriormente, el *frontend* se ha implementado de manera que el usuario no puede introducir valores fuera de rango.

CAPÍTULO 8

ESCENARIO DE EJEMPLO

8. ESCENARIO DE EJEMPLO

Una vez mostrado y conocido el funcionamiento teórico de todos los aspectos necesarios para la implementación final es el momento de realizar un escenario de ejemplo para comprobar el correcto funcionamiento del proyecto en general, tanto del levantamiento del clúster como de la funcionalidad de la aplicación creada.

Para ello se hará un recorrido a través de los distintos elementos, desde los microservicios a encapsular hasta el lanzamiento de peticiones a través de la aplicación creada pasando por la creación y configuración del clúster.

8.1. EJECUTANDO MICROSERVICIOS EXISTENTES

Los microservicios de prueba son aplicaciones muy simples, únicamente muestra cada una un mensaje *"Hello World"* acompañado de un número según el puerto en el que estén.

Se han creado los servicios de prueba en Eclipse haciendo uso de *Spring Boot*, un *framework* el cual ahorra tiempo de trabajo ya que, entre otra cosas, permite implementar un servidor *Tomcat* sobre el cual lanzar las aplicaciones. Al haber tres aplicaciones diferentes que tienen que lanzarse en un mismo host el primer paso será modificar la configuración del servidor para que cada una de las aplicaciones utilice un puerto distinto, siendo los elegidos 8080, 8081 y 8082.

Los servicios tienen un código Java similar al mostrado en la Figura 61 donde se muestra el directorio raíz de la aplicación lanzada en el puerto 8080, es decir, al acceder a **localhost:8080** aparecerá el mensaje que indica las rutas hacia los otros servicios.

```java
@RequestMapping(value = "/", method = RequestMethod.GET)
public ResponseEntity<String> main(){
    ResponseEntity<String> re = new ResponseEntity<String>(
            "Página por defecto: "
            + "8080/hello = Hello World "
            + "8081/hello = Hello World 1 "
            + "8082/hello = Hello World 2 ", HttpStatus.OK);
    return re;
}
```

Figura 61: Código Java de un microservicio

Ejecutando las tres aplicaciones en el IDE se puede acceder a las distintas rutas y ver los mensajes correspondientes según el puerto. Como aparece en la Figura 62 cada ruta está lanzada por una aplicación distinta que a su vez tiene su propia configuración para crear el servidor. De esta manera se puede acceder a todas las aplicaciones de manera simultánea.

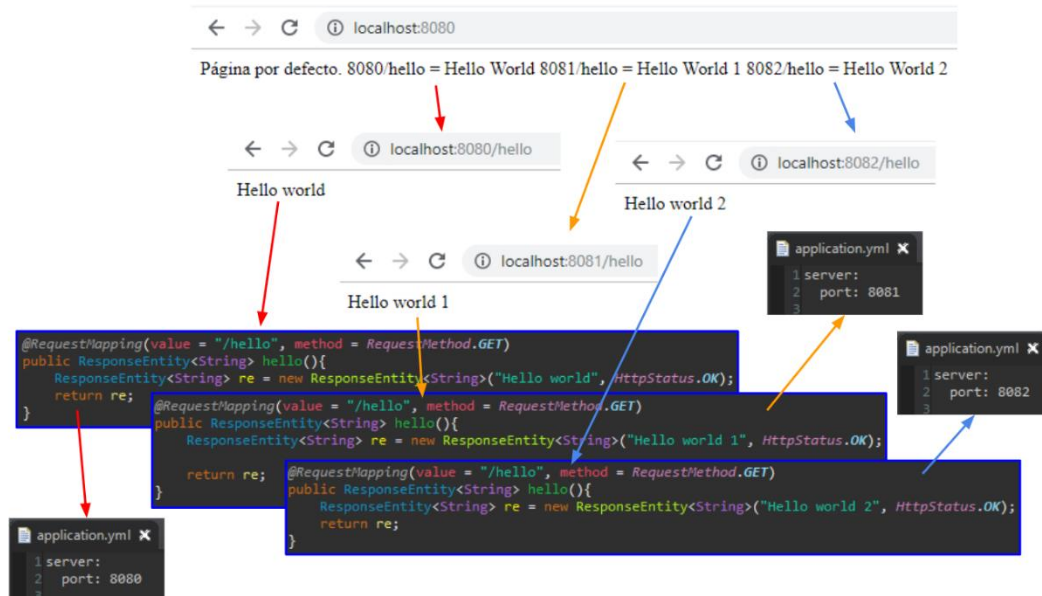

Figura 62: Relación entre configuración y rutas de acceso

Hasta este punto, aunque las aplicaciones son simples, también lo es su forma de ejecución desde Eclipse ya que con la configuración actual, gracias a *Spring Boot,* se crea un servidor Tomcat donde ejecutar las aplicaciones, como se muestra en la Figura 63. Pero sin el IDE de intermediario el proceso sería un poco diferente y la ejecución no sería tan simple y sencilla. Para solucionar este problema haremos uso de *Docker* donde simplemente se encapsula la aplicación en un contenedor y se ejecuta.

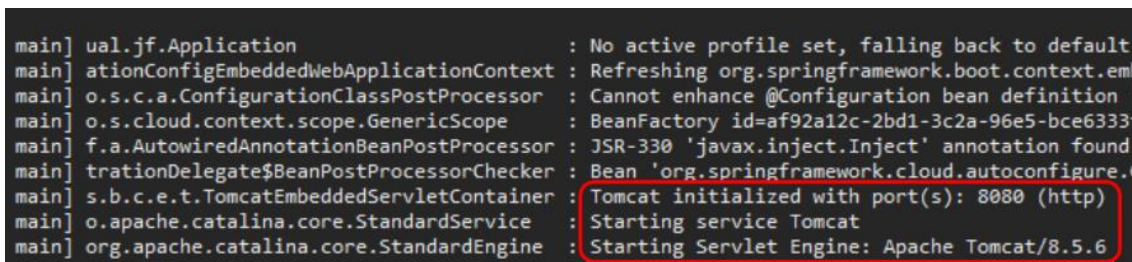

Figura 63: Inicio de Tomcat por parte de SpringBoot

8.2. ENCAPSULACIÓN EN CONTENEDORES

Una vez tenemos las aplicaciones a encapsular, el primer paso es crear una imagen para el contenedor de cada una de ellas. Esta imagen se creará a partir de un *Dockerfile* y deberá tener los comandos necesarios para ejecutar una aplicación Java.

Para obtener el archivo **.jar** se hará uso de **Maven** mediante el comando `mvn clean package` en la carpeta donde se encuentre el archivo **pom.xml**.

Con los archivos generados se crea un *Dockerfile* con el siguiente contenido:

```
FROM openjdk:alpine

COPY ./Application-0.0.1-SNAPSHOT.jar /usr/src/application/

WORKDIR /usr/src/application

ENTRYPOINT ["java", "-jar", "Application-0.0.1-SNAPSHOT.jar"]
```

Así, al construir la imagen:

1. Se descarga la imagen *jdk* si no se encuentra en local. La versión *alpine* es la más básica por lo que ocupará menos espacio en disco.
2. Se copia el archivo *.jar* a la ruta indicada dentro del contenedor
3. Se establece como directorio de trabajo el lugar donde se encuentra la aplicación
4. Se establece el comando a ejecutar cuando se cree el contenedor, que es el necesario para ejecutar la aplicación Java

Por lo que el contenedor que ejecute la imagen seguirá los pasos necesarios para ejecutar también la aplicación que contiene. Para que un contenedor pueda almacenar la imagen descrita en el *Dockerfile* es necesario primero construirla mediante el comando `docker build`. Además, por comodidad, se le añadirá una etiqueta, quedando de la forma:

```
$ docker build -t app1 .
```

Siguiendo este método se crean los archivos *Dockerfile* para las tres aplicaciones, así como su construcción cambiando en cada uno de ellos el nombre de la imagen o el archivo a ejecutar dentro del contenedor. Una vez construidas todas las imágenes se pueden observar la lista de las mismas con el comando `docker images` (Figura 64)

```
root@ubuntu:/# docker images
REPOSITORY                      TAG            IMAGE ID
app2                            latest         8cb9912f28cf
app3                            latest         8cb9912f28cf
app1                            latest         837882727250
```

Figura 64: Imágenes de Docker existentes en local

El siguiente paso es levantar contenedores que ejecuten las imágenes que acabamos de crear con el comando `docker run` indicando el puerto a exponer y el nombre de la imagen para cada contenedor.

```
$ docker run -p 8080:8080 app1
```

Con este comando se mapea el puerto 8080 del contenedor (el puerto donde se ejecuta Tomcat con la aplicación) con el puerto 8080 del host.

Al lanzar la aplicación en el IDE anteriormente existía el problema que el puerto 8080 estaba ocupado por una aplicación, sin embargo, ahora que se ejecutan en contenedores diferentes todas las aplicaciones pueden utilizar el puerto por defecto de *Tomcat* y funcionará correctamente siempre y cuando el puerto del host al que se mapean no sea el mismo. Si no se cambia la configuración inicial hay que exportar el puerto 8080 para la aplicación 1, el puerto 8081 para la 2 y el puerto 8082 para la tercera aplicación.

```
$ docker run -p 8081:8081 app1
```

```
$ docker run -p 8083:8082 app1
```

Levantados los contenedores se puede acceder a las rutas correspondientes y observar la salida correcta (Figura 65).

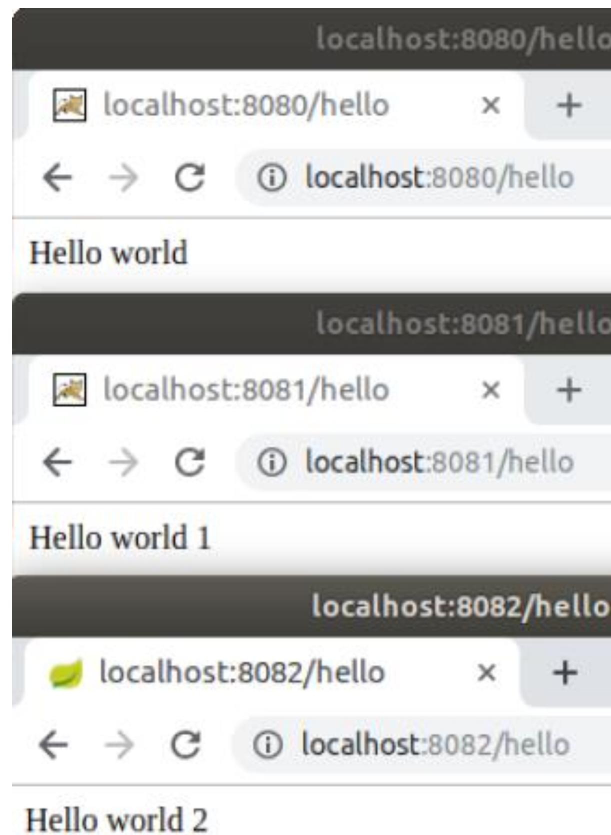

Figura 65: Rutas de acceso a los microservicios

Hasta ahora todo funciona correctamente pero el proceso es repetitivo y tedioso. Para simplificar y agilizar el levantamiento haremos uso de *docker-compose*.

8.2.1. UTILIZANDO DOCKER-COMPOSE

Gracias a *docker-compose* podemos ejecutar múltiples aplicaciones en contenedores simultáneamente. En este caso queremos levantar las tres aplicaciones anteriores sin tener que realizar un *Dockerfile* para cada una de ellas, además de ejecutar los comandos para levantar cada contenedor de forma independiente.

Creando un archivo *docker-compose.yaml*, similar al que se muestra a continuación, se obtienen los resultados necesarios.

```
version: '3'

services:

  app1:

    image: app1

    container_name: app1-container

    network_mode: "host"

    expose:

      - "8080"
```

```
app2:
 image: app2
 container_name: app2-container
 network_mode: "host"
 expose:
   - "8081"

app3:
 image: app3
 container_name: app3-container
 network_mode: "host"
 expose:
   - "8082"
```

En el archivo se indica para cada uno de los servicios a crear, nombrados como **app1**, **app2** y **app3** la imagen a ejecutar en el contenedor, que es la creada anteriormente, el nombre del contenedor a levantar y la red a utilizar junto con el puerto a exponer. De esta manera accediendo a los puertos 8080, 8081 y 8082 de localhost se acceden a los diferentes contenedores.

Con el comando `docker-compose up` se iniciará el proceso de creación de los contenedores, lo que mostrará por pantalla de manera simultánea el flujo de salida de los tres (Figura 66).

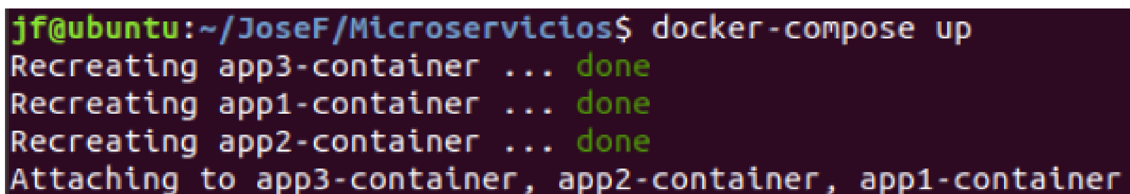

Figura 66: Creación de contenedores con docker-compose

8.3. ORQUESTANDO EN KUBERNETES: MINIKUBE

En este punto tenemos la aplicación configurada para ejecutarla con un único comando. Podría considerarse que el avance ha sido grande pero no es todo. En el momento en que una aplicación multiservicio se está ejecutando necesitamos controlar los contenedores en los cuales se ejecuta. Esto no es una tarea sencilla y *Docker* proporciona algunas herramientas, pero no las suficientes. Se quiere que la aplicación se controle y se escale según distintos requisitos a introducir (que se caiga un servicio, que reciba muchas peticiones…) de manera automática. Para ello haremos uso de *Kubernetes*.

Como se ha explicado anteriormente la manera más sencilla de tener un clúster de *Kubernetes* en ejecución en un entorno local es utilizando *Minikube*. Teniendo *Minikube* configurado para comunicarse con la API de *Kubernetes* el proceso a seguir es similar al que se ha realizado mientras se trabaja únicamente con *Docker*: crear los objetos necesarios para la ejecución de los servicios.

Mediante el comando `minikube start` se crea el clúster de *Kubernetes* y, ya que se está ejecutando en una máquina virtual, se hace uso del parámetro `--vm-driver=none`

Al estar en un entorno y con una herramienta distinta a *Docker* el proceso no será exactamente igual pero sí similar. El primer paso es levantar los contenedores, en este caso los despliegues.

De igual manera que anteriormente se han utilizado archivos *Dockerfile* en este caso se hará uso de archivos *.yaml* que contienen los datos necesarios para que la API de *Kubernetes* lo transforme en objetos del clúster. Los archivos de configuración de los despliegues son similares al que se muestra a continuación

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1
spec:
  selector:
    matchLabels:
      app: app1
  template:
    metadata:
      labels:
        app: app1
    spec:
      containers:
      - name: app1-container
        image: app1
        ports:
        - containerPort: 8080
```

Donde se especifica la versión de la API que traducirá el archivo, el tipo de objeto que es (despliegue) y los datos necesarios para crear el objeto, tanto el nombre como las especificaciones.

El campo `selector` indica cómo el despliegue a crear encuentra el *pod* con el que está relacionado para ser administrado. Es este caso mediante la concordancia de etiquetas `app1` que se define más abajo. Por último, se configura la red para tener el puerto 8080 accesible ya que es el necesario para comunicarnos con el programa.

Una vez configurado el archivo, mediante el comando `apply` se inicia el proceso para generar el objeto en el clúster.

```
root@ubuntu:/# kubectl apply -f app1-deployment.yaml
```

Si observamos los despliegues y *pods* creados se puede comprobar si se ha iniciado correctamente o no, para ello haremos uso de los comandos `get` de *Kubernetes*:

```
root@ubuntu:/# kubectl get po
```

```
root@ubuntu:/# kubectl get deployment
```

La respuesta de los comandos es la siguiente (Figura 67):

```
root@ubuntu:/# kubectl get deployment
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
app1    0/1     1            0           4m
root@ubuntu:/# kubectl get po
NAME                    READY   STATUS             RESTARTS   AGE
app1-755c57967-9vtft    0/1     ImagePullBackOff   0          4m30s
```

Figura 67: Despliegues y pods en el clúster

Se puede observar que hay un error, más concretamente **ImagePullBackOff** que impide que se inicie el despliegue, por eso se muestra que 0 de 1 están listos. Este error es debido a que *Kubernetes* no puede encontrar la imagen indicada. En el archivo de configuración del despliegue se ha indicado la imagen creada anteriormente llamada **app1**, ¿por qué *Docker* sí la encontraba y *Kubernetes* no?

La respuesta es sencilla (ver Figura 68). Para buscar una imagen primero se hace a nivel local y si no se encuentra se busca en el repositorio de la nube indicado (si no se indica ninguno se utiliza DockerHub por defecto). La imagen creada anteriormente está en local accesible para *Docker* pero ya que *Minikube* inicia un clúster en una máquina virtual con *Kubernetes* instalado no tiene acceso a esas imágenes ya que busca en el registro dentro de la máquina virtual creada. Por el contrario, si no hace uso de máquina virtual el motor de *Docker* lo configura a parte del existente, por lo que los archivos que contenga, como las imágenes, no son accesibles.

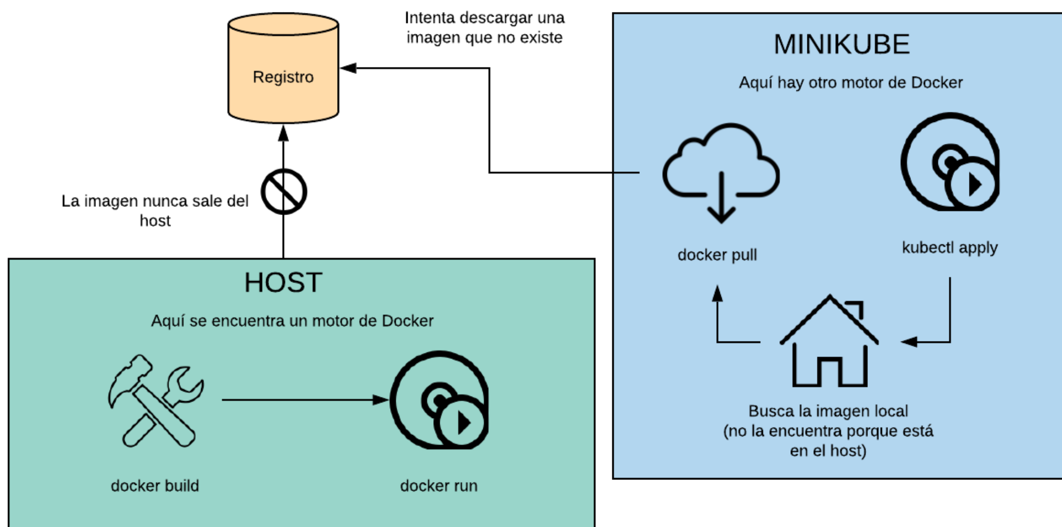

Figura 68: Problema presente al intentar utilizar una imagen construida en local desde Minikube

Para solucionar este problema existen varias soluciones, siendo la más sencilla subir la imagen a un repositorio en la nube y descargándola cada vez que sea necesario, pero es una opción que requiere de conexión a internet y no siempre es un recurso disponible por lo que, teniendo la imagen en el

mismo equipo en el que se trabaja, es más adecuado encontrar una solución que permita compartir registros entre *Docker* en el host y *Docker* en *Minikube*.

La opción más rápida es configurar *Minikube* para que apunte al entorno de *Docker* mediante el comando `eval $(minikube docker-env)` y asegurarse que el despliegue se inicialice con el parámetro `imagePullPolicy: Never` para que no intente descargar la imagen ya que no la encontrará. Después de esto es necesario volver a construir las imágenes en el nuevo entorno y al ejecutar el despliegue funcionará correctamente.

Sin embargo, si no se utiliza ningún driver para la inicialización de *Minikube* el proceso será mucho más sencillo ya que únicamente hay que indicar en el archivo de creación que no se descargue nunca la imagen, ya que el paso anterior de configurar los entornos para hacer que *Minikube* apunte al de *Docker* es lo que se realiza por defecto para poder ejecutarse sin drivers. Así que únicamente sería necesario realizar la construcción con *Minikube* iniciado.

Como factor a tener en cuenta, en cuanto se configure el entorno para poder acceder a la imagen, *Kubernetes* creará el *pod* y lo ejecutará sin necesidad de reiniciarlo ya que ante el fallo el *pod* se mantiene en un proceso en el que intenta obtener la imagen de forma indefinida, así en cuanto lo consigue se levanta el contenedor aunque necesite reiniciarse varias veces hasta conseguirlo, como se muestra en la Figura 69 donde el número de reinicios necesarios han sido seis.

```
root@ubuntu:/# kubectl get deployment
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
app1   1/1     1            1           6m24s
root@ubuntu:/# kubectl get po
NAME                    READY   STATUS    RESTARTS   AGE
app1-6996989794-xkcgt   1/1     Running   6          6m27s
```

Figura 69: Despliegues y pods existentes en el clúster funcionando

Solucionado el problema podemos crear los despliegues en el clúster. En un mismo archivo se pueden insertar tantos objetos como se deseen. Así que, siguiendo la estructura expuesta anteriormente, se pueden crear todos los despliegues. Además, el comando `apply` levantará los objetos indicados en el archivo que no estén en el clúster mientras que si ya se encuentran activos buscará si la configuración es diferente para modificarlo o para dejarlo como está. Es una gran ventaja frente al comando `créate`, ya que con éste, si el objeto indicado en el archivo existe en el clúster dará un error al no poder crear dos iguales.

Para crear todos los objetos en un mismo archivo deben ir separados por “---“ y seguir la misma estructura que si fuese un archivo con un único objeto. Por ejemplo, si el archivo para desplegar la **app1** es el indicado anteriormente (añadiendo `imagePullPolicy: Never)` para crear el despliegue de las otras dos aplicaciones sería:

```
apiVersion: apps/v1
      . . . → archivo app1
---
apiVersion: apps/v1
      . . . → archivo app2
```

```
---
apiVersion: apps/v1
      . . . → archivo app3
```

Una vez creados los despliegues con sus *pods* correspondientes, para comprobar que se han levantado correctamente la mejor manera es accediendo al contenido. Para ello será necesario utilizar un servicio. Como se explica anteriormente el servicio desempeña un papel similar a un intermediario entre el despliegue y su acceso de forma remota, así pues creando un servicio enlazado al despliegue se podrá acceder al contenedor a través de una URL. El comando utilizado es el siguiente:

```
$ kubectl expose deployment/app1 --type="NodePort" --port 8080
```

Donde se especifica la aplicación que se quiere hacer accesible, la manera de hacerlo (`NodePort` lo hace accesible fuera del clúster, pero para que sea accesible únicamente dentro sería necesario utilizar `ClusterIP`) y el puerto el cual conecta con el exterior del clúster, en este caso `8080` que es donde se ejecuta la aplicación.

Al introducir el comando la consola mostrará un mensaje (Figura 70) notificando que se ha expuesto el *pod* correctamente.

```
root@ubuntu:/# kubectl expose deployment/app1 --type="NodePort" --port 8080
service/app1 exposed
```

Figura 70: Creación de un servicio mediante línea de comandos

En este punto sólo falta conocer la dirección IP del servicio creado para poder acceder a él y, por tanto, a la aplicación. Es importante recordar que al crear un clúster no se accede a él desde **localhost** (a menos que se haga uso de un proxy) sino desde los objetos o las APIs configuradas para ello. Para conocer la IP a la que se quiere acceder una manera sencilla es listando todos los servicios existentes, lo que mostrará su dirección. Además, *Minikube* facilita este paso ya que con el comando `minikube service <nombre_servicio> --url` muestra directamente la dirección IP asociada al servicio.

Para listar los servicios se hace uso de `kubectl get svc` y para comprobar el correcto funcionamiento se hace **curl** a la ruta indicada, como se muestra en la Figura 71.

```
root@ubuntu:~# kubectl get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
app1         NodePort    10.107.79.108   <none>        8080:30038/TCP   5m37s
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP          21m
root@ubuntu:~# curl $(minikube service app1 --url)/hello
Hello worldroot@ubuntu:~#
```

Figura 71: Acceso al servicio creado en el clúster

Accediendo a la ruta **/hello** muestra el mensaje *"Hello world"* lo cual indica que el servicio se ha levantado correctamente.

Otra manera de crear el servicio es mediante un archivo *.yaml* que siga la estructura mostrada a continuación:

```
apiVersion: v1
kind: Service
metadata:
  name: app1
  labels:
    app: app1
spec:
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      nodePort: 30038
      protocol: TCP
  selector:
    app: app1
```

Con esta configuración se indica el tipo de objeto que se quiere crear, `Service`, el nombre que tendrá, `app1`, el tipo, `NodePort`, y los puertos que se enlazan `8080 - 30038` del *pod* que coincida con la etiqueta definida en `selector`.

De esta manera, siguiendo la misma estructura con la que anteriormente se definieron tres despliegues en un mismo archivo, se pueden definir tres servicios para crearlos simultáneamente.

8.3.1. ESCALANDO APLICACIONES: HPA

Con las aplicaciones en funcionamiento, cada una de manera independiente, existe la posibilidad de que queramos escalar una o varias de ellas. *Kubernetes* ofrece la posibilidad de realizar el escalado manualmente mediante comandos o a través de un objeto (HPA) definido en un archivo *.yaml*.

El primer requisito para iniciar el escalado según los recursos es indicar los límites de los mismos al clúster, es decir, comunicarle al clúster la cantidad de memoria o CPU máxima para más adelante indicar a partir de qué porcentaje se quiere iniciar el escalado. Para ello se deben añadir nuevos campos en la definición del despliegue, estos son los referentes al atributo `resources` mostrados a continuación.

```
apiVersion: apps/v1
kind: Deployment
...
```

```
spec:
  ...
  template:
  ...
    spec:
      containers:
      ...
      resources:
        requests:
          memory: 50m
```

En este ejemplo se indica que la cantidad de memoria disponible para el despliegue es de 50MB. Aplicando el comando `kubectl apply -f archivo.yaml` se configura el despliegue sin necesidad de tener que eliminarlo. Una vez configurado el despliegue podemos definir el auto escalador para que funcione según estás métricas.

La parte fundamental de un archivo que define un HPA son los atributos del campo `spec` ya que ahí se indica el objeto que se quiere escalar, la cantidad de réplicas, tanto máximas como mínimas, el tipo de métrica que se utiliza para decidir el escalado y la cantidad. La definición viene dada por la siguiente estructura.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: app1-hpa-resources
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: app1
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
```

```
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Donde este archivo indica que:

- Se crea un objeto de tipo `HorizontalPodAutoscaler` haciendo uso de la API `autoscaling/v2beta2` de *Kubernetes* con nombre `app1-hpa-resources`
- Se especifica que se quiere escalar un objeto `Deployment` llamado `app1` que debe tener entre 2 y 10 réplicas.
- La métrica utilizada para decidir el escalado es un recurso (`Resource`) de nombre `cpu` que actúa cuando se utiliza (`Utilization`) más del 50%

Una vez configurado el escalador se puede crear el objeto y comprobar si funciona correctamente haciendo uso del comando `kubectl describe hpa app1-hpa-resources` observando los mensajes que muestra por pantalla. Si todo funciona correctamente se podrá leer entre los eventos el mensaje `the HPA was able to successfully calculate a replica count from cpu resource utilization (percentage of request)`, mientras que el objeto creado estará leyendo las métricas y mostrando el porcentaje de uso, como se muestra en la Figura 72.

```
root@ubuntu:/# kubectl get hpa
NAME                 REFERENCE         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
app1-hpa-resources   Deployment/app1   0%/50%    2         10        2          54s
```

Figura 72: HPA existente en el clúster

El porcentaje de CPU o memoria utilizados puede variar según la cantidad de peticiones concurrentes que reciba y por tanto las operaciones que tiene que realizar. Al ser una aplicación muy simple la que se está monitorizando ya que únicamente debe mostrar un mensaje, la cantidad de peticiones deberá ser elevada para consumir recursos de manera considerable. Si la acción que realiza la aplicación fuese más pesada como calcular decimales del número PI serían necesarias menos peticiones para un mayor uso de los recursos.

En la Figura 73 se muestra cómo se realiza el escalado de dos a tres *pods* debido a que el HPA detecta que el porcentaje de recursos utilizados está por encima del indicado.

```
From                       Message
----                       -------
horizontal-pod-autoscaler  New size: 2; reason: Current number of replicas below Spec.MinReplicas
horizontal-pod-autoscaler  New size: 3; reason: cpu resource utilization (percentage of request) above target
```

Figura 73: Mensajes de escalado del clúster

8.3.2. UTILIZANDO PROMETHEUS

Una vez realizado escalado con memoria y CPU, es el turno de las métricas personalizadas. Para ello el primer paso es generar los objetos de *Prometheus* necesarios: el operador y el adaptador.

El operador se hará cargo únicamente de permitir a *Prometheus* monitorizar el clúster y generar sus propias métricas, mientras que el adaptador le permitirá intercambiar esa información con el clúster para que éste actúe según los valores leídos.

Para disponer del adaptador en el clúster primero creamos un *namespace* de nombre “monitoring” para incluir todos los objetos ahí siguiendo buenas prácticas. De esta manera en caso de querer eliminar *Prometheus* del clúster únicamente sería necesario eliminar el *namespace* **monitoring**.

Seguidamente se crea la configuración que se va a añadir al *pod* y el propio *pod*. De esta manera prometheus se encuentra activo en el clúster pudiendo acceder a él desde el interior, como se muestra en la Figura 74.

```
root@ubuntu:/#kubectl get svc -n monitoring
NAME         TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
prometheus   NodePort   10.110.182.144   <none>        9090:32138/TCP   3m56s
root@ubuntu:/#minikube ssh
                         _             _
            _         _ ( )           ( )
  ___ ___  (_)  ___  (_)| |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| || , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)

$ curl 10.110.182.144:9090
<a href="/graph">Found</a>.
```

Figura 74: Acceso al servicio desde el interior de Minikube

El problema es que para tener más detalles lo ideal es poder acceder a la interfaz de usuario que incorpora, para ello a través del servicio creado se accede a la IP externa generada por *Minikube* mediante la cual se puede acceder desde fuera del clúster (Figura 75).

```
root@ubuntu:/#curl $(minikube service prometheus -n monitoring --url)
<a href="/graph">Found</a>.

root@Random:/home# 
```

Figura 75: Acceso al servicio desde el exterior de Minikube

En este momento, desde el punto de vista del auto escalador no se ha generado ningún cambio, ya que, al igual que antes, no puede acceder a métricas que no sean uso de memoria o CPU. Si intentamos escalar según alguna de las muchas métricas provistas por *Prometheus, Kubernetes* aceptará la configuración como válida y se generará un autoescalador, pero éste nunca conseguirá su objetivo.

Para la lectura de métricas personalizadas debe añadirse al archivo de configuración del HPA los siguientes atributos dentro del campo de especificaciones:

```
metrics:
- type: Pods
  pods:
    metricName: http_requests
    targetAverageValue: 10
```

Donde se especifica que la métrica leída de un *pod* se llama `http_requests` y que actuará cuando la cantidad llegue a 10. Esto quiere decir que en el momento en el que el *pod* reciba más de 10 peticiones por segundo el HPA creará una nueva réplica.

Una vez creado, haciendo uso del comando `describe` se muestra el siguiente error (Figura 76)

```
  Warning  FailedGetPodsMetric          5s (x4 over 50s)  horizontal-pod-autos
caler  unable to get metric http_requests: unable to fetch metrics from custom
metrics API: no custom metrics API (custom.metrics.k8s.io) registered
```

Figura 76: Fallo del HPA al intentar hacer uso de la API de métricas personalizadas

Como se explica anteriormente, esto es debido a que la API de *Kubernetes* no es capaz de leer el dato resultante de la métrica **http_requests** que facilita *Prometheus*. Para conseguir la comunicación entre ambas partes se hace uso de un adaptador.

Para levantar el adaptador se compilan en el clúster los archivos ofrecidos por el repositorio oficial de *Kubernetes* siguiendo las instrucciones explicadas previamente. En el momento en el que *Prometheus* encuentre el *pod* estará disponible para leer la métrica **http_requests**.

Como se muestra en la Figura 77, tras la configuración del clúster, el HPA se encuentra disponible para escalar el recurso según la métrica personalizada

```
Message
-------
recommended size matches current size
the HPA was able to successfully calculate a replica count from pods metric http_requests
the desired count is within the acceptable range
```

Figura 77: Mensaje del HPA confirmando la lectura de la métrica personalizada

La columna *target* indica la cantidad de peticiones indicadas como máximo cada segundo. La cantidad de peticiones actuales se indica en "mili-unidades", en este caso "mili-peticiones", es decir, que será necesario que el valor llegue a 10.000 para que el HPA cree una nueva réplica. Como muestra de ello se puede comprobar en la Figura 78 como el escalado pasa de 2 a 3 cuando se supera el valor umbral.

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
app1-hpa	Deployment/app1	2627m/10	2	10	2	25m
app1-hpa	Deployment/app1	4635m/10	2	10	2	25m
app1-hpa	Deployment/app1	15110m/10	2	10	2	27m
app1-hpa	Deployment/app1	16656m/10	2	10	4	27m
app1-hpa	Deployment/app1	17943m/10	2	10	4	28m
app1-hpa	Deployment/app1	18252m/10	2	10	4	28m
app1-hpa	Deployment/app1	12186m/10	2	10	4	28m
app1-hpa	Deployment/app1	9182m/10	2	10	4	29m
app1-hpa	Deployment/app1	9202m/10	2	10	4	29m
app1-hpa	Deployment/app1	9227m/10	2	10	4	29m
app1-hpa	Deployment/app1	9248m/10	2	10	4	29m

Figura 78: Replicado de pods según la carga recibida

Es cierto que no es inmediato que al superar el valor límite se cree una nueva réplica porque se contabiliza a partir de que está disponible. Es decir, una vez que se llega a 10.000 el clúster inicia el proceso para crear una nueva réplica y ese proceso tarda unos segundos hasta que el nuevo *pod* está operativo. Es por eso que con valor 15110 todavía existen únicamente 2 réplicas, pero justo después, en menos de un minuto, el número total se incrementa pasando a ser 4.

8.4. INTERFAZ WEB

Con el correcto funcionamiento del clúster, el siguiente paso es hacer uso de la interfaz web para realizar las peticiones necesarias sobre los objetos elegidos. Con el clúster iniciado, al acceder a la ruta de *Tomcat* en el puerto 8080 aparece la ventana de inicio de la aplicación en la que se pueden elegir los distintos objetos. En este caso haremos un envío de peticiones al servicio **app1** y veremos cómo varían los despliegues, así como el auto escalador asociado.

En la interfaz principal elegiremos los objetos que queremos controlar. Todos ellos serán referentes a nuestra aplicación creado **app1**. En cuanto a las peticiones los campos se pueden rellenar o dejar vacíos, en cuyo caso se pondrían los valores por defectos de *Hey*. Como ejemplo se ha decidido dejar un campo vacío e introducir valores en los otros dos, tal y como se muestra en la Figura 79.

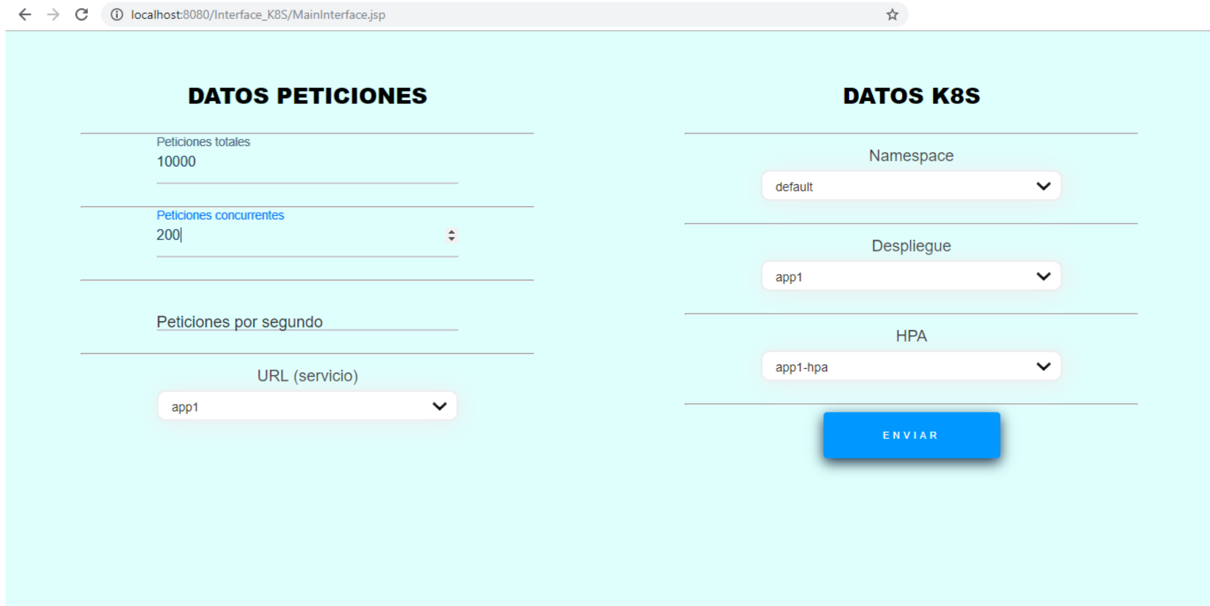

Figura 79: Interfaz web. Pantalla principal

Realizada la configuración, pulsando el botón "Enviar" se accede a la ventana donde se muestran los resultados mientras se realizan las tareas en un segundo plano.

Durante el proceso de envío de peticiones se puede observar como varían los diferentes datos tanto del HPA como de despliegues activos (Figura 80).

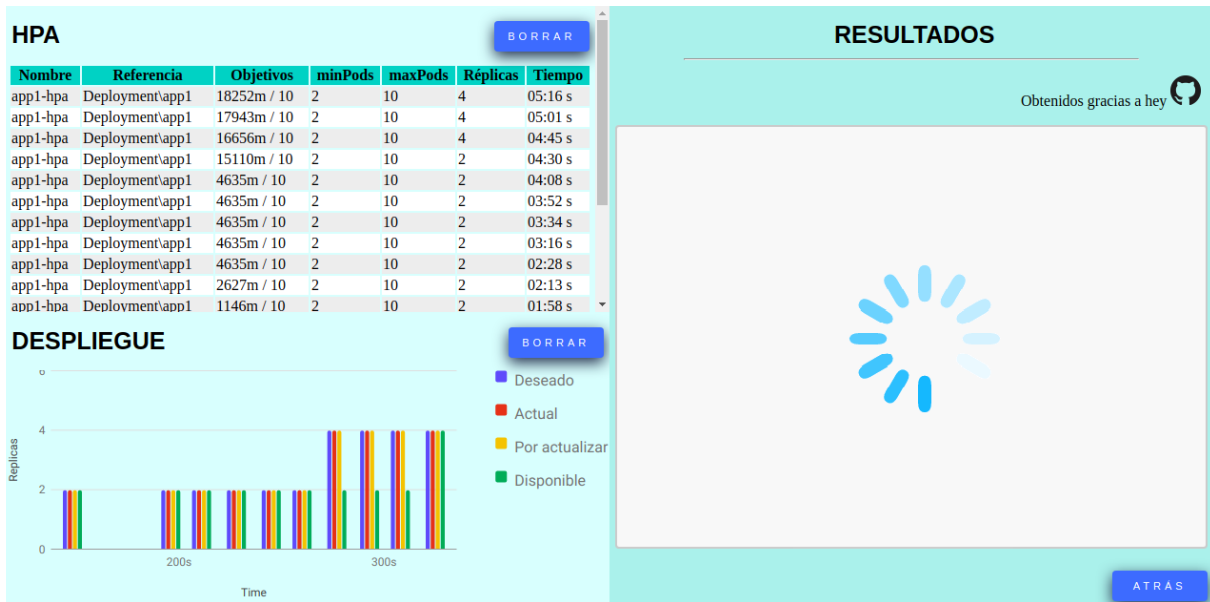

Figura 80: Interfaz web. Envío de peticiones

Y una vez que el programa termina de realizar las peticiones se muestran los resultados obtenidos a la derecha en la interfaz. De este modo podemos obtener en una única interfaz tres tipos de datos distintos: auto escalador, despliegues y envío de peticiones. Lo que ahorra tener que estar con, mínimo, tres terminales abiertos para observar los mismos resultados.

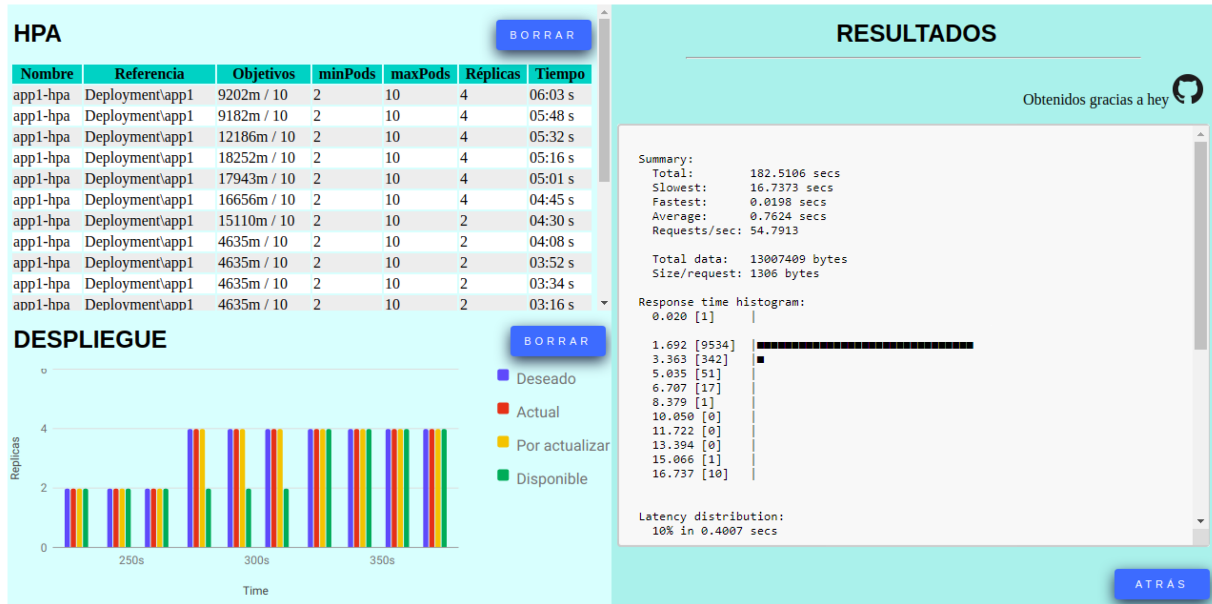

Figura 81: Interfaz web. Muestra de resultados

CAPÍTULO 9

CONCLUSIONES

9. CONCLUSIONES

La tecnología avanza conforme avanza el tiempo y no adaptarse es un problema de gran calibre para cualquier empresa. Cada día los usuarios son más exigentes y la competencia mayor, por lo que un segundo fuera de servicio puede suponer la pérdida de dinero y reputación.

A lo largo de este documento se han tratado distintos temas de gran interés a nivel de negocio, desde la concepción de la arquitectura que a día de hoy cuenta con más apoyo hasta la visualización del escalado de los distintos contenedores de un clúster mediante una interfaz web. Todos ellos tienen un punto en común, intentar mejorar el proceso de desarrollo y entrega de software.

Con el crecimiento de usuarios de nuevas tecnologías año tras año se hace imprescindible pensar en una arquitectura acorde a los requisitos de los clientes. Un cambio de la tecnología utilizada incluye un cambio en la entrega del software; donde la arquitectura monolítica tenía un único servicio la arquitectura de microservicios tiene varios. Es por eso que el uso de Docker supone una mejora a nivel empresarial al facilitar múltiples tareas repetitivas y propensas a fallos. Además, permite mantener un entorno aislado, ya sea para desarrollo, pruebas o simplemente por seguridad.

Se ha mostrado como el uso de contenedores mejora la eficiencia de la empresa y es útil para todas las partes interesadas en el proyecto. No es únicamente una herramienta para entregar software de forma más rápida y eficiente, sino también para crear un entorno de desarrollo acorde a las necesidades. Permite probar cualquier aplicación sin dañar al equipo o probar nuevas funcionalidades. Sea cual sea el propósito, Docker facilita el proceso.

Debido a la creación de múltiples contenedores es necesario tener una manera de gestionarlos. Es por eso por lo que se hace uso de una herramienta que permita organizar los contenedores en un clúster como es *Kubernetes*. Con los contenedores organizados y fácilmente manejables las oportunidades a nivel de negocio aumentan ya que se disponen de un mejor entorno de trabajo y más eficiente. Además, con el funcionamiento interno de la propia herramienta se subsana uno de los principales problemas de las empresas a día de hoy: dejar de emitir servicio.

Ya que *Kubernetes* permite el escalado de aplicaciones y asegura que siempre haya un mínimo de procesos indicados en ejecución, una empresa podría estar tranquila sabiendo que su aplicación seguirá estando operativa, aunque se produzca un fallo. Y no solo el escalado es una ventaja para la empresa, también lo es la monitorización de sus servicios.

Existen diversos motivos por los que querer controlar un servicio. Ya sea para asegurar el correcto funcionamiento o para conocer el uso que se le da, a nivel de negocio es un factor clave. Hoy en día cualquier empresa que tenga un producto en el mercado quiere saber los máximos detalles posibles sobre éste. La monitorización permite conocer el estado de cada uno de ellos y controlarlo gracias a métricas personalizadas que pueda crear la empresa para su uso.

En este documento se han aprovechado las métricas para juntar dos conceptos como son escalado y monitorización y así modificar la cantidad de servicios disponibles según lo solicitado que estuviese, ya que tener un clúster que aumenta el número de aplicaciones disponibles, en un contexto donde el ataque más común es por denegación de servicio distribuidos (*DDoS*), puede suponer mantener un servicio disponible.

Se ha creado un escenario donde una aplicación de microservicios recibía peticiones masivas y no ha dejado de ser funcional. Todo ello con una configuración casi automatizada que podría incluirse en un script y ejecutarla mediante un único comando.

Cualquier empresa querría asegurar eficiencia, facilidad de uso, protección del equipo, monitorización, automatización a la hora de levantar servicios caídos y replicar ya sea por métricas propias o recursos, entre otros factores, con la ejecución de un único script. Con las herramientas presentadas ha sido posible gracias a la manera en que la tecnología basada en contenedores ha conseguido acoplar distintas funcionalidades software, lo que supone un aumento de calidad para cualquier tecnología.

Por último, se ha creado una interfaz web para conocer el estado del clúster mientras recibe una gran carga de peticiones. Con esto se ha intentado ahorrar tener que realizarlo desde tres lugares distintos ya que con pulsar un solo botón se envían y muestran los datos necesarios.

Por todo lo explicado anteriormente se considera que lo expuesto en este documento podría ayudar a mejorar el desarrollo y entrega del software ya sea a nivel empresarial o personal.

REFERENCIAS BIBLIOGRÁFICAS

REFERENCIAS BIBLIOGRÁFICAS

- [1] Chapter 1: Service Oriented Architecture (SOA). (2019). Recuperado de https://web.archive.org/web/20160206132542/https://msdn.microsoft.com/en-us/library/bb833022.aspx . ...pág. 6

- [2] What Is SOA?. (2019). Recuperado de https://web.archive.org/web/20160819141303/http://opengroup.org/soa/source-book/soa/soa.htm...pág. 7

- [3] Lewis, J., & Fowler, M. (2019). Microservices. Recuperado de https://martinfowler.com/articles/microservices.html ....pág. 7

- [4] Microservices vs. SOA - DZone Microservices. (2019). Recuperado de https://dzone.com/articles/microservices-vs-soa-2 .... pág. 8

- [5] Microsoft - Microservices architecture style. (2018). Recuperado de https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices .... pág. 8

- [6] What is container (containerization or container-based virtualization)? - Definition from WhatIs.com. (2019). Recuperado de https://searchitoperations.techtarget.com/definition/container-containerization-or-container-based-virtualization ....pág. 9

- [7] Nickoloff, J., (2016). *Docker in Action*, Shelter Island, NY: Manning Publications Co. ..... pág. 9

- [8] Docker overview. (2019). Recuperado de https://docs.docker.com/engine/docker-overview/ .... pág. 10

- [9] Docker Engine (2019). Arquitectura de Docker. Recuperado de https://docs.docker.com/engine/docker-overview/ .... pág. 10

- [10] Erl, T. (2004). *Service-Oriented Architecture*. East Rutherford, NJ: Prentice Hall PTR. pág. 11

- [11] Goasguen, S. (2015). *Docker Cookbook.* Sebastopol, CA: O'Reilly Media, Inc. .... pág. 11

- [12] Kubernetes Documentation. (2019). Recuperado de https://kubernetes.io/docs/home/ . ...pág. 11

- [13] Saito, H., Lee, H., & Wu, C. (2017). *DevOps with Kubernetes*. Birmingham, Reino Unido: Packt Publishing Ltd. ....pág. 12

- [14] Picking the Right Solution. (2019). Recuperado de https://v1-12.docs.kubernetes.io/docs/setup/pick-right-solution/ .... pág. 12

- [15] Docker overview. (2019). Recuperado de https://docs.docker.com/engine/docker-overview/ .... pág. 21

- [16] Use the Docker command line. (2019). Recuperado de https://docs.docker.com/engine/reference/commandline/cli/ .... pág. 25

- [17] Networking Overview. (2019). Recuperado de https://docs.docker.com/network/ .. pág. 27

- [18] Saito, H., Lee, H., & Wu, C. (2017). *DevOps with Kubernetes*. Birmingham, Reino Unido: Packt Publishing Ltd. .... pág. 27

- [19] Dockerfile Reference (2019). Recuperado de https://docs.docker.com/engine/reference/builder/ .... pág. 28

- [20] What is Container Orchestration?. (2019). Recuperado de https://avinetworks.com/glossary/container-orchestration/ .... pág. 31

- [21] Docker Compose. (2019). Recuperado de https://docs.docker.com/compose/ .... pág. 32

- [22] Mouat, A. (2015). *Using Docker*. Sebastopol, CA: O'Reilly Media Inc. .... pág. 34

- [23] Kubernetes Components. (2019). Recuperado de https://kubernetes.io/docs/concepts/overview/components/ .... pág. 36

- [24] Saito, H., Lee, H., & Wu, C. (2017). *DevOps with Kubernetes*. Birmingham, Reino Unido: Packt Publishing Ltd. .... pág. 39

- [25] Kubernetes Components. (2019). Recuperado de https://kubernetes.io/docs/concepts/overview/components/ .... pág. 41

- [26] Kubernetes, Minikube. (2019). Recuperado de https://github.com/kubernetes/minikube ....pág. 44

- [27] Kubernetes, Kompose. (2019). Recuperado de https://github.com/kubernetes/kompose ....pág. 45

- [28] User Guide. (2019). Recuperado de http://kompose.io/user-guide/ .... pág. 46

- [29] Cluster Networking. (2019). Recuperado de https://kubernetes.io/docs/concepts/cluster-administration/networking/ .... pág. 47

- [30] Saito, H., Lee, H., & Wu, C. (2017). DevOps with Kubernetes. Birmingham, Reino Unido: Packt Publishing Ltd. .... pág. 48

- [31] What is Monitoring Software?. (2019). Recuperado de https://www.techopedia.com/definition/4313/monitoring-software .... pág. 50

- [32] Web UI (Dashboard). (2019). Recuperado de https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/ .... pág. 54

- [33] Overview | Prometheus. (2019). Recuperado de https://prometheus.io/docs/introduction/overview/ .... pág. 56

- [34] Prometheus – Architecture overview. (2019). Recuperado de https://github.com/prometheus/prometheus .... pág. 56

- [35] Saito, H., Lee, H., & Wu, C. (2017). DevOps with Kubernetes. Birmingham, Reino Unido: Packt Publishing Ltd. .... pág. 56

- [36] Storage | Prometheus. (2019). Recuperado de https://prometheus.io/docs/prometheus/latest/storage/ .... pág. 57

- [37] Alertmanager | Prometheus. (2019). Recuperado de https://prometheus.io/docs/alerting/alertmanager/ .... pág. 57

- [38] Prometheus Operator (2019). Recuperado de https://github.com/coreos/prometheus-operator .... pág. 57

- [39] Perform Rolling Update Using a Replication Controller. (2019). Recuperado de https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/ ....pág. 65

- [40] Horizontal Pod Autoscaler. (2019). Retrieved from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ .... pág. 66

- [41] K8S Prometheus Adapter Configuration. (2019). Recuperado de https://github.com/DirectXMan12/k8s-prometheus-adapter/blob/master/docs/config-walkthrough.md .... pág. 68

- [42] Hey. (2019). Recuperado de https://github.com/rakyll/hey .... pág. 70

- [43] Charts | Google Developers. (2019). Recuperado de https://developers.google.com/chart/ .... pág. 84

ANEXO A:

RECURSOS ADICIONALES

ANEXO A: RECURSOS ADICIONALES

1. Dockerfile de la aplicación 1.

app1.dockerfile

```
1  FROM openjdk:alpine
2  COPY ./app3.jar /usr/src/app/
3  WORKDIR /usr/src/app
4  ENTRYPOINT ["java", "-jar", "app3.jar"]
5
```

2. Dockerfile de la aplicación 2.

app2.dockerfile

```
1  FROM openjdk:alpine
2  COPY ./app2.jar /usr/src/app/
3  WORKDIR /usr/src/app
4  ENTRYPOINT ["java", "-jar", "app2.jar"]
5
```

3. Dockerfile de la aplicación 3.

app3.dockerfile

```
1  FROM openjdk:alpine
2  COPY ./app3.jar /usr/src/app/
3  WORKDIR /usr/src/app
4  ENTRYPOINT ["java", "-jar", "app3.jar"]
```

4. Configuración del despliegue de la aplicación 1.

app1-deployment.yml

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: app1
spec:
  selector:
    matchLabels:
      app: app1
  template:
    metadata:
      labels:
        app: app1
      annotations:
        prometheus.io/scrape: 'true'
    spec:
      containers:
      - name: app1
        image: app1
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
        resources:
          requests:
            memory: "256Mi"
            cpu: "100m"
          limits:
            memory: "512Mi"
            cpu: "200m"

```

5. Configuración del despliegue de la aplicación 2.

app2-deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app2
spec:
  selector:
    matchLabels:
      app: app2
  template:
    metadata:
      labels:
        app: app2
    spec:
      containers:
      - name: app2-container
        image: app2
        imagePullPolicy: Never
        ports:
        - containerPort: 8081
        resources:
          requests:
            memory: "256Mi"
            cpu: "100m"
          limits:
            memory: "512Mi"
            cpu: "200m"
```

6. Configuración del despliegue de la aplicación 3.

! app3-deployment.yml ×

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app3
spec:
  selector:
    matchLabels:
      app: app3
  template:
    metadata:
      labels:
        app: app3
    spec:
      containers:
      - name: app3-container
        image: app3
        imagePullPolicy: Never
        ports:
        - containerPort: 8082

```

7. Configuración del servicio de la aplicación 1.

! service-app1.yml ×

```yaml
apiVersion: v1
kind: Service
metadata:
  name: app1
  labels:
    app: app1
spec:
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      nodePort: 30038
      protocol: TCP
  selector:
    app: app1

```

8. Configuración del servicio de la aplicación 2.

! service-app2.yml ×

```yaml
apiVersion: v1
kind: Service
metadata:
  name: app2
  labels:
    app: app2
spec:
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      nodePort: 30039
      protocol: TCP
  selector:
    app: app2

```

9. Configuración del servicio de la aplicación 3.

! service-app3.yml ×

```yaml
apiVersion: v1
kind: Service
metadata:
  name: app3
  labels:
    app: app3
spec:
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      nodePort: 30040
      protocol: TCP
  selector:
    app: app3

```

10. Configuración del HPA utilizando métricas basadas en recursos.

hpa-app2-resource-metrics.yml

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: app2-hpa-resources
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: app1
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

11. Configuración del HPA utilizando métricas personalizadas.

hpa-app1-custom-metrics.yml

```yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: app1-hpa
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: app1
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Pods
    pods:
      metricName: http_requests
      targetAverageValue: 10
```

12. Configuración del namespace “monitoring”

namespace-monitoring.yml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: monitoring

```

13. Script para automatizar todo el proceso.

script.sh

```bash
#Apuntar al entorno de Minikube
eval $(minikube docker-env)

#Construir imágenes de las APPs
docker build -t app1 ./Docker/app1/
docker build -t app2 ./Docker/app2/
docker build -t app3 ./Docker/app3/

#Prometheus
kubectl apply -f Prometheus/Operator/

#Generar certificados
make certs

#Custom metrics API
kubectl apply -f Prometheus/Adapter/

#Levantar la aplicación en el clúster
kubectl apply -f APP

```

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

El desarrollo de software utilizando una arquitectura basada en microservicios ha aumentado en los últimos años siendo adoptado por grandes empresas como Amazon, LinkedIn, Netflix o SoundCloud ya que propone notables mejoras frente a las arquitecturas monolíticas. La arquitectura de microservicios consiste en un conjunto de servicios relativamente pequeños y autónomos que trabajan juntos comunicándose entre ellos, se modelan para un mismo objetivo y tienen un propósito único y claramente definido.

Uno de los factores del crecimiento de la arquitectura de microservicios se debe al reciente aumento en popularidad de las tecnologías basadas en contenedores y en orquestación de los mismos, lo que facilita la implementación y la hace más viable desde el punto de vista técnico.

Las tecnologías basadas en contenedores, como por ejemplo Docker, facilitan el despliegue de las aplicaciones. Docker es un software que ofrece la capacidad de empaquetar y ejecutar una aplicación en un entorno aislado, automatizando así el despliegue, permitiendo además ejecutar múltiples contenedores en un mismo host.

Para la orquestación de dichos contenedores existen distintos proyectos como *Docker Swarm*, o *Kubernetes*, siendo este último el referente actual. *Kubernetes* es un sistema utilizado para la automatización del despliegue, escalado y manejo de aplicaciones desplegadas en contenedores.

El conjunto de las distintas tecnologías definen el marco hacia donde se dirige el desarrollo de software tanto en la actualidad como en el futuro próximo.

GRADO EN INGENIERÍA INFORMÁTICA, 2018/2019