Universidad de Extremadura

Escuela Politécnica

Grado en Ingeniería Informática
en Ingeniería de Computadores

Trabajo Fin de Grado

KubeKVM App

Pablo Moralo Flores

Julio 2019

Universidad de Extremadura

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado

KubeKVM App

Autor: D. Pablo Moralo Flores

Fdo:

Tutor: D. Antonio Manuel Silva Luengo

Fdo:

Tribunal Calificador

Presidente: D.

Secretario: D.

Vocal: D.

CALIFICACIÓN:

Julio, 2019.

Índice general

Índice de figuras

"Inténtalo, vuelve a fallar, falla mejor."
- Samuel Beckett -

Abstract

This document contains all the work done to obtain this final application called KubeKVM App, which implements a REST API developed in Node.js and Express, which is connected to a MongoDB database where the data inserted in the REST API is stored, that is packaged inside a Docker container and that is supported on a Kubernetes cluster. In it the teacher enters the necessary data for the deployment of a number of virtual machines KVM configured for their subject. The cluster has a daemon running a script developed in Python for reading the data and its subsequent deployment using KVM according to the needs of the subject taught by the teacher.

To reach this software, a study and analysis of current virtualization technologies is carried out. After this analysis it was necessary to have storage in the cluster of the University of Extremadura, both for the deployment of the Kubernetes cluster where the REST API is hosted and for the deployment of the relevant virtual machines at each moment.

Keywords:
Docker, Express, Kubernetes, KVM, MongoDB, Node.js, Python, REST API, Virtualization.

Resumen

En este documento se recoge todo el trabajo realizado para obtener esta aplicación final denominada KubeKVM App, que implementa una API REST desarrollada en Node.js y Express, que esta conectada con una base de datos MongoDB donde se almacenan los datos insertados en la API REST, que se encuentra empaquetada dentro de un contenedor Docker y que está apoyada sobre un clúster de Kubernetes. En ella el docente introduce los datos necesarios para el despliegue de cierto número de maquinas virtuales KVM configuradas para su asignatura. El clúster cuenta con un daemon corriendo un script desarrollado en Python para la lectura de los datos y su posterior despliegue mediante KVM en función de las necesidades de la asignatura impartida por el docente.

Para llegar a dicho software, se realiza un estudio y análisis de las tecnologías actuales de virtualización. Tras dicho análisis fue necesario contar con almacenamiento en el clúster de la Universidad de Extremadura, tanto para el despliegue del cluster de Kubernetes donde se aloja la API REST como para el despliegue de las maquinas virtuales pertinentes en cada momento.

Capítulo 1

Introducción

El gran avance de la sociedad y de las tecnologías ligadas a ella, nos obliga a estudiar y analizar la necesidad que hay hoy en día de proporcionar un servicio rápido, eficiente y con alta disponibilidad a la gran cantidad de usuarios consumidores de servicios en la red. Para esto es necesario que nuestros servicios cuenten con un alto grado de escalabilidad para que no haya retardos en el envío y recepción de la información, un alto grado de disponibilidad para que los datos siempre este accesibles, y un alto grado de automatización para que los sistemas que hospedan los servicios creados tengan capacidad de autoregeneración ante fallos. Ante esta situación se plantean dos soluciones conjuntas, encapsular y tratar los servicios mediante el uso de contenedores software Docker y Kubernetes.

Pero, ¿qué es un contenedor software? Son paquetes de elementos que permiten ejecutar una aplicación determinada en cualquier sistema operativo. Se utilizan para garantizar que una determinada aplicación se ejecute correctamente cuando cambie su entorno, sin dar fallos de ningún tipo.

Y, ¿qué es Kubernetes? Es una plataforma de código abierto, extensible y portátil para administrar cargas de trabajo y servicios en contenedores, que facilita tanto la configuración declarativa como la automatización. Proporciona un entorno de gestión centrado en contenedores. Organiza la infraestructura de computación, redes y almacenamiento en nombre de las cargas de trabajo de los usuarios. Esto proporciona gran parte de la simplicidad de la Plataforma como un Servicio (PaaS) con la flexibilidad de la Infraestructura como un Servicio (IaaS), y permite la portabilidad entre los proveedores de infraestructura.

Debido a todas estas ventajas y la facilidad para implementarlo, estas tecnologías se han situado como objetivo para suplir las deficiencias de sistemas utilizados en producción en el ámbito laboral por falta de flexibilidad, portabilidad, escalabilidad, disponibilidad o automatización de procesos dentro del mismo.

1.1. Objetivos

El motivo principal por el cual se ha desarrollado este proyecto es el de facilitar a los docentes de la *Escuela Politécnica de Cáceres* un servicio que proporcione una herramienta para el despliegue de las máquinas necesarias a los alumnos para el desarrollo de los laboratorios de las asignaturas impartidas por los mismos.

Por lo tanto, los objetivos que se querían conseguir eran:

- Utilizar **Kubernetes** para la orquestación de los contenedores pertenecientes a los servicios ofrecidos para satisfacer las necesidades de los despliegues de cada asignatura impartida por los docentes. Esta tecnología debe proporcionar una manera sencilla de despliegue de aplicaciones y servicios junto a la facilidad de gestión de los mismos, pero sobre todo el objetivo principal es proveer a la aplicación de una alta disponibilidad y escalabilidad.

- Desarrollar una **API** que sirva la información referente a los despliegues solicitados, el estado de los mismos y el resto de datos que se encuentran en la base de datos. Este servicio debe proporcionar una forma sencilla de acceder a los datos, es decir, facilitando las consultas y peticiones al servidor mediante el envío de mensajes solicitando los datos necesarios, algo que proporciona *REST*. Esta **aplicación Web** también ofrece la información de las base de datos forma gráfica y permita interactuar con el usuario.

- Implementar un **script** para consumir los datos de la API REST con el fin de realizar la creación y despliegue de las máquinas especificadas mediante el uso de la virtualización.

- Virtualizar las maquinas especificadas en los despliegues mediante la tecnología **KVM** basada en la técnica de virtualización total utilizando un sistema de hipervisor.

A partir de los objetivos principales anteriores también surgen otros de importancia para dotar de funcionalidad al software final. Estos objetivos son:

- Implementar una **REST** para la comunicación con el servicio ya que es un protocolo basado en patrones de intercambio de mensajes que utiliza directamente HTTP para obtener datos o indicar la ejecución de operaciones en cualquier formato sin las abstracciones adicionales de este tipo de protocolos.

- Desarrollar un sistema de consultas **CRUD** para facilitar la petición de los datos a la base de datos, para que el usuario realice las consultas de manera sencilla o para facilitar el acceso a los datos al desarrollador que manipule la API sin necesidad de utilizar la url.

- Diseñar una **Web** que represente la información almacenada en la base de datos para que el usuario consulte la información necesaria mediante el envío de mensajes a la REST, estos mensajes serán consultas a nuestra API REST la cual es una aplicación que cuenta con un con un conjunto de consultas CRUD(Create/Read/Update/Delete), que utiliza todos los verbos HTTP.

1.2. Antecedentes/Estado del arte

En el siguiente apartado se exponen las tecnologías estudiadas para iniciar el proyecto, su finalidad, uso, necesidades y el aporte que estas podían realizar para alcanzar el objetivo final.

Entre todas estás tecnologías, las dos decisiones mas importantes a realizar eran en base a la virtualización de las maquinas pertenecientes a cada despliegue solicitado sobre la aplicación:

- Inicialmente se realizó un estudio a fondo sobre las diferentes implementaciones de APIS de virtualización para Kubernetes realizadas por la comunidad con tecnologías como KubeVirt, Virtlet, RancherVM o Kata Containers. Finalmente se desecharon todas al no ser posible en ninguna de ellas la opción de introducir dentro de un contenedor una máquina virtual configurada para el desarrollo de los laboratorios de una asignatura, también fue un factor importante la poca persistencia que tienen los contenedores por definición ya que están creados para ser elementos efímeros los cuales no tienen sentido al necesitar unas máquinas configuradas en la que los alumnos puedan trabajar y guardar sus datos.

- Una vez tomada la decisión anterior, se debía elegir que tipo de tecnología de virtualización iba a ser utilizada. Actualmente hay una gran cantidad de programas con licencia gratuita como: KVM, LXC, OpenVZ(última version de LXC), Proxmox, VirtualBox, Xen, VMWare, Al querer usar un sistema de hipervisor, un tipo de virtualización total y ser el sistema anfitrión Linux, la elección final fue KVM, ya que al ser este un módulo mas de Linux no hace falta modificar el kernel, lo podemos cargar y descargar en tiempo de ejecución, soporta los demás módulos, viene por defecto en el kernel, funciona en todo tipo de máquinas, servidores, escritorio o laptop, permite la migración en caliente de máquinas virtuales y ejecutar múltiples máquinas virtuales cada una con su propia instancia, y ademas hereda ventajas que ya tiene el kernel como escalabilidad, soporte para diferentes procesadores, gestion de memoria y acceso a memoria no uniforme(NUMA).

1.2.1. MongoDB

MongoDB es una es un sistema de base de datos NoSQL orientado a documentos, esto significa que está constituida por un conjunto de programas que almacenan, recuperan y gestionan datos de documentos o datos de algún modo estructurados o semiestructurados.

Estas bases de datos de documentos son intuitivas y suelen utilizarse ya que los datos en el nivel de la aplicación generalmente se representan como un documento JSON. Pueden conservarse datos utilizando el mismo formato de modelo de documento que se usa en el código de la aplicación. En ellas, cada documento puede tener la misma estructura de datos o no, y cada documento es autodescriptivo, incluyendo su posible esquema único, y no depende necesariamente de ningún otro documento. Los documentos se agrupan en colecciones, que tienen un propósito similar al de una tabla en una base de datos relacional.

Además MongoDB esta diseñado para aplicaciones web y aplicaciones de infraestructura de internet, ya que soporta gran cantidad de usuarios mediante el uso de escalabilidad horizontal, es decir, MongoDB permite crear un clúster donde la información estará particionada y replicada lo que permite escalar nuestro sistema horizontalmente en base a la demanda.

Con respecto al entorno de la aplicación desarrollada, esta base de datos almacena la información sobre el tipo de despliegue solicitado por el docente con datos como el nombre del despliegue, tipo, estado y número de máquinas a desplegar.

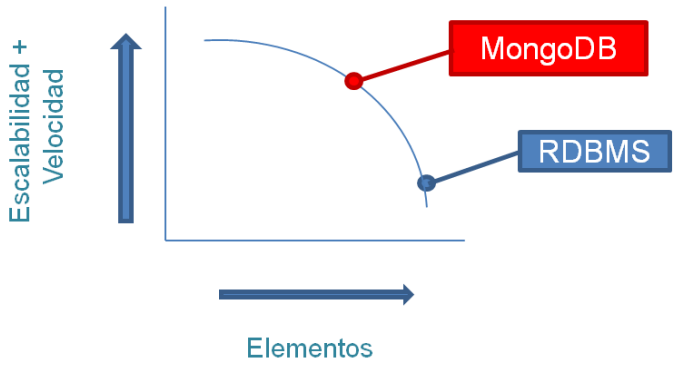


Figura 1.1: Rendimiento MongoDB frente a RDBMS.

Observando la figura, podemos ver como MongoDB se encuentra en la zona óptima, en la que la velocidad y la escalabilidad son altas, así como también es el número de objetos de la base de datos. Por contra en una base de datos relacional (RDBMS) tanto la escalabilidad como la velocidad se ven penalizados ante un número elevado de elementos en la base de datos.

1.2.2. Node.js

Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google. Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables, como por ejemplo, servidores web.

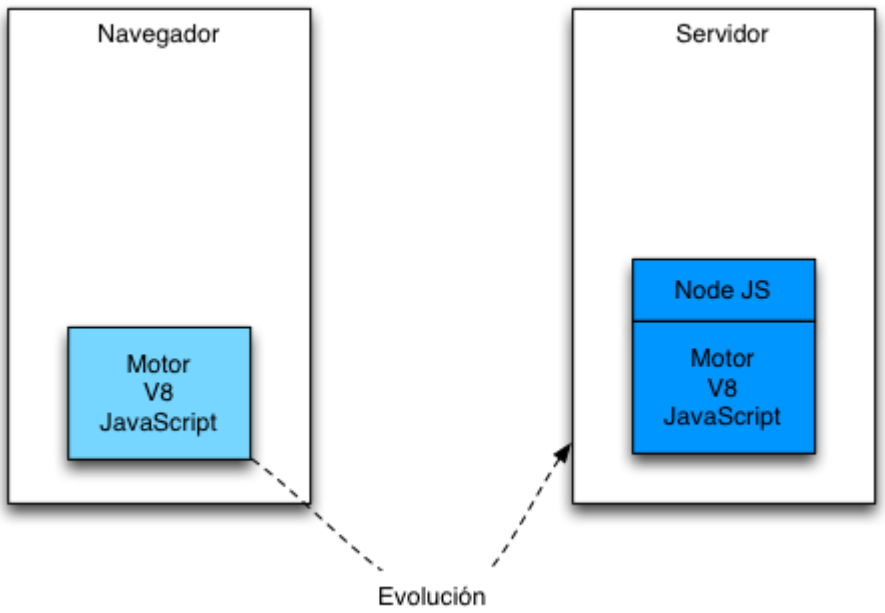


Figura 1.2: El motor V8 de Google.

Node.js esta basado en el motor V8 de Javascript de Google. Este motor está diseñado para correr en un navegador y ejecutar el código de Javascript de una forma extremadamente rápida. La tecnología que está detrás de Node.js permite ejecutar este motor en el lado del servidor abriendo un nuevo abanico de posibilidades en cuanto al mundo de desarrollo se refiere.

También se han instalado módulos adicionales para agregar funcionalidades como el framework para Node denominado Express, Method Override para extender la funcionalidad de los formularios y Mongoose para permitir la conexión con MongoDB.

1.2.3. REST

La Transferencia de Estado Representacional o REST se usa para describir cualquier interfaz entre sistemas que utilice directamente HTTP para obtener datos o indicar la ejecución de operaciones sobre los datos, en cualquier formato, sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes.

Con la necesidad de implementar un sitio web moderno para que pueda ser consumido desde una aplicación web, se llegó a la conclusión de construirlo mediante una API RESTful, que es aquella API que emplea todos los verbos HTTP(GET,POST,PUT y DELETE)

Por tanto se procedió al desarrollo de un servidor web que sirva una API RESTful usando para ello la tecnología de Node.js y MongoDB como base de datos. Este conjunto también suele ser denominado MEAN Stack, el cual es un framework o conjunto de subsistemas de software para el desarrollo de aplicaciones, y páginas web dinámicas, que están basadas en el lenguaje de programación JavaScript. Gracias a esta característica el conjunto se integra exitosamente en una plataforma autosuficiente.

Figura 1.3: Desarrollo basado en API REST.

En la figura anterior vemos que lo importante es que el cliente no recibe HTML para renderizar, sino simplemente los datos que se han generado como respuesta. Es decir, el servidor no escribe HTML, sino únicamente genera los datos para enviarlos al cliente.

1.2.4. Docker

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos, es decir, la idea detrás de Docker es crear contenedores ligeros y portables para las aplicaciones software que puedan ejecutarse en cualquier máquina con Docker instalado, independientemente del sistema operativo que la máquina tenga por debajo, facilitando así también los despliegues.

El principal motivo de la elección Docker es que permite meter en un contenedor todas aquellas cosas que una aplicación necesita para ser ejecutada y la propia aplicación. Así puede llevarse ese contenedor a cualquier máquina que tenga instalado Docker y que se ejecute la aplicación sin tener que hacer nada más, ni preocuparse de qué versiones de software tiene instalada esa máquina, de si tiene los elementos necesarios para que funcione la aplicación, de si son compatibles...

Por tanto, Docker nos permite poder centrarnos en desarrollar el código de la aplicación sin preocuparse de si dicho código funcionará en la máquina en la que se ejecutará.

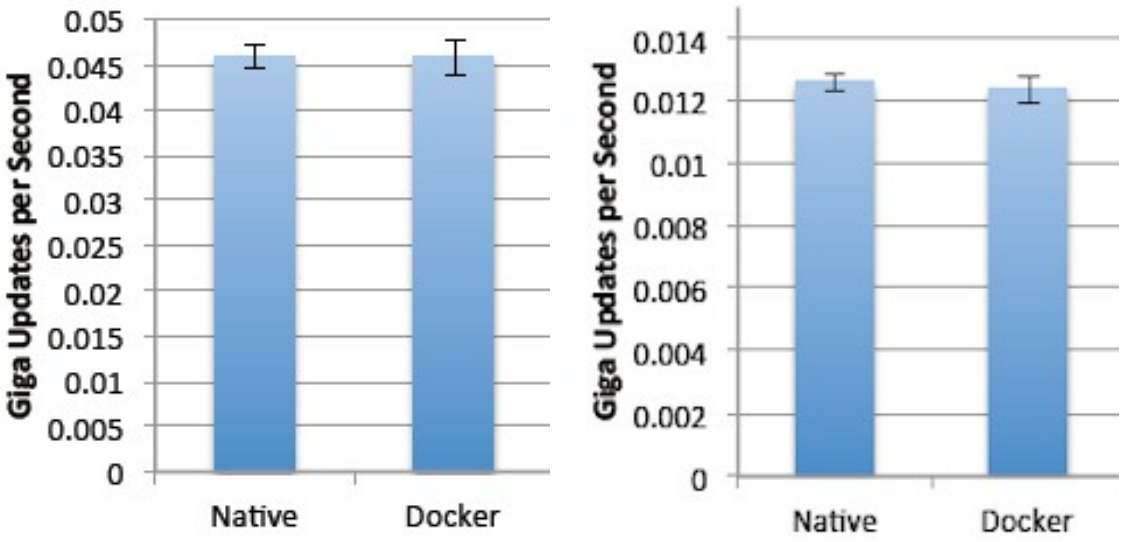


Figura 1.4: Acceso aleatorio y almacenamiento E/S de Docker y un servidor físico.

Podemos observar que tanto en el acceso aleatorio como en una prueba de almacenamiento donde se realizan lecturas y escrituras secuenciales, el rendimiento de Docker esta a la par con un servidor físico.

1.2.5. Kubernetes

Kubernetes es un sistema de código libre para la automatización del despliegue, ajuste de escala y manejo de aplicaciones en contenedores que fue originalmente diseñado por Google y donado a la Cloud Native Computing Foundation.

A la hora de realizar el despliegue de nuestra aplicación, elegimos realizar la instalación de Kubernetes sobre el clúster de la universidad por las ventajas que esta tecnología nos ofrece como la orquestación de contenedores en múltiples hosts, el fácil escalado tanto a nivel lógico como a nivel físico, la posibilidad de escalar en número de contenedores que se están ejecutando, el poder optimizar recursos y la automatización sobre despliegues que permite.

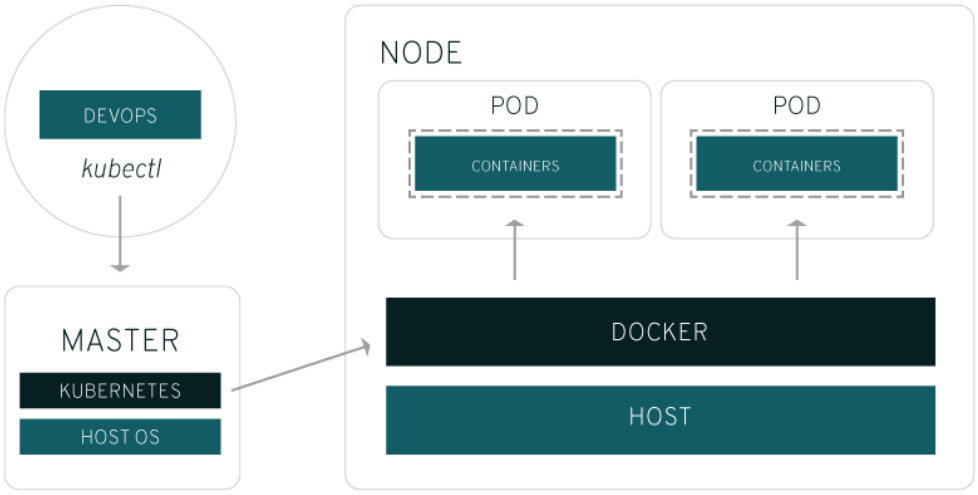


Figura 1.5: Adaptación de Kubernetes a infraestructura.

El diagrama muestra como desde el punto de vista de la infraestructura, hay un cambio muy pequeño en la manera de administrar los contenedores. El control sobre esos contenedores comienza en un nivel superior, lo que le ofrece un mejor control sin la necesidad de microadministrar cada contenedor o nodo por separado. Aunque es necesario realizar algunas tareas como asignar un master de Kubernetes, definir nodos y definir pods.

1.2.6. KVM

Kernel-based Virtual Machine o KVM, es una solución para implementar virtualización completa con Linux. Está formada por un módulo del núcleo y herramientas en el espacio de usuario, siendo en su totalidad software libre. El componente KVM para el núcleo está incluido en Linux desde la versión 2.6.20.

KVM permite ejecutar máquinas virtuales utilizando imágenes de disco que contienen sistemas operativos sin modificar. Cada máquina virtual tiene su propio hardware virtualizado: una tarjeta de red, discos duros, tarjeta gráfica, etc.

Como ya fue comentado anteriormente, se utilizó la tecnología de virtualización KVM por el principal motivo de querer usar un sistema de hipervisor, un tipo de virtualización total y ser el sistema anfitrión Linux.

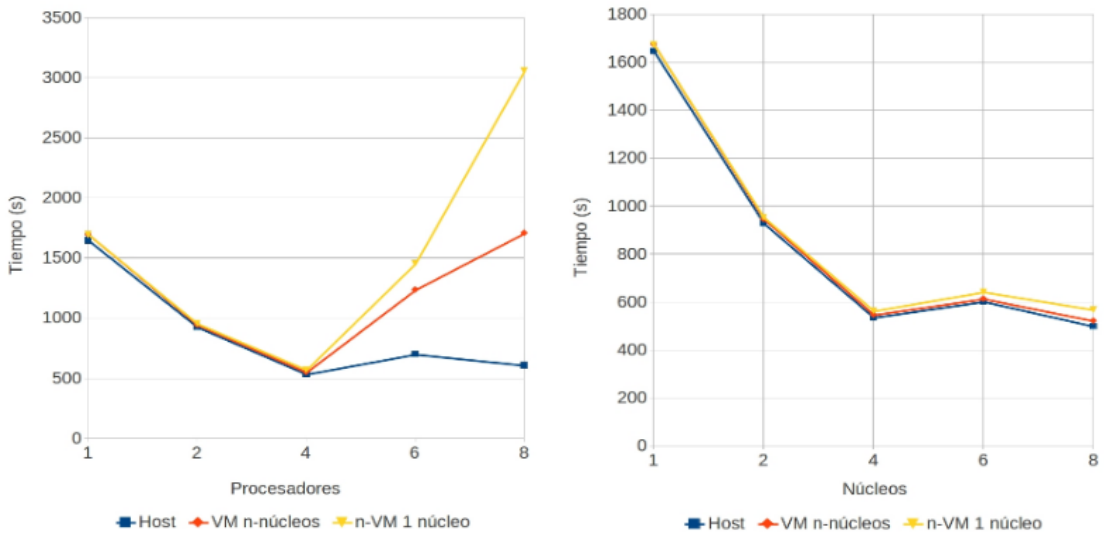


Figura 1.6: Tiempos de ejecución sin y con hyperthreading.

En la figura de la izquierda vemos que los tiempos de ejecución al emplear un número de procesos inferior o igual al núumero de cores disponibles en el microprocesador produce resultados similares en los tipos de máquinas empleadas. Sin embargo, si se aumenta el número de cores hasta que se supera el número de cores del host anfitrión, se pone de manifiesto un deterioro del rendimiento que se hace muy notable en el caso de las máquinas virtualizadas. En cambio en la figura de la derecha observamos que los resultados obtenidos con el hypertheading activado ponen de manifiesto una mejora sustancial del rendimiento cuando el número de cores físicos del anfitrión se supera, mostrando un resultado muy similar en los tipos de máquinas empleadas.

Capítulo 2

Análisis y diseño

En este apartado se realiza un análisis del problema y el desarrollo del diseño del software, definiendo que es lo necesario y su coincidencia con los objetivos marcados para la obtención del producto final.

2.1. Requisitos

A continuación se realiza de definición de los requisitos, facilitando una descripción del comportamiento del software

2.1.1. Requisitos funcionales

- Proporcionar la web al usuario desde la petición a un puerto de cualquiera de los pods de Kubernetes que exponen ese servicio.
- Permitir al usuario crear la conexión con el servidor y el cierre de la misma por cualquiera de las 2 partes.
- Se debe ofrecer al usuario un punto de acceso para la inserción de despliegues. Cuando se inserta un despliegue se debe especificar el nombre, tipo y el número de réplicas.
- El usuario debe tener permisos para gestionar los despliegues de manera que pueda actualizarlos o eliminarlos.
- Informar al usuario que ocurre con sus mensajes a la hora de la creación, consulta, actualización o eliminación de despliegues.
- Los diferentes puntos de acceso deben incluir el código HTTP correspondiente en la respuesta, en función de si la petición ha sido procesada con éxito o hay algún error.
- Cuando un usuario inicia sesión en la plataforma, debe encontrar una pantalla de inicio donde pueda elegir entre diferentes pestañas para realizar un test de balanceo de carga o realizar acciones referentes a los despliegues como crearlos, modificarlos o eliminarlos.
- Los diferentes despliegues creados serán mostrados junto con la información asociada a ellos: nombre, tipo, replicas y estado. En dicha página, el usuario tendrá soporte para poder actualizar o eliminar esos despliegues.

- Si el usuario elige el paso de crear un despliegue cuando está dentro de la plataforma, se le debe mostrar un formulario donde poder insertar los datos del misma: nombre, tipo, replicas y estado.
- Permitir acceso a los datos de la base de datos de forma parcial mediante el uso del servicio para la consulta de los despliegues existentes.
- Proporcionar al usuario un servicio automatizado de alta disponibilidad mediante las diferentes herramientas con las que cuenta Kubernetes. En un principio no se prevé un gran volumen de tráfico, pero se debe diseñar una infraestructura donde pueda crecer el número de peticiones y no suponga ningún problema de rendimiento.
- Creación de máquinas virtualizadas con las imágenes especificadas por el usuario en las peticiones al servicio.

2.1.2. Requisitos no funcionales

- No permitir consultas directas a la base de datos.
- El tiempo de aprendizaje del usuario debe ser menor a 2 horas.
- Se deben ofrecer mensajes de error cuando sea necesario, de manera que estos informen al usuario final sobre el error producido.
- Debe tener interoperabilidad, es decir, debe ofrecer acceso por parte de otras aplicaciones.
- Permitir escalabilidad y controlarla mediante el controlador de replicación.
- Realizar balanceo de carga en los nodos que ofrecen el servicio.
- Ofrecer la posibilidad de una fácil y sencilla modificación y/o ampliación del software por parte del desarrollador.
- Uso del modelo de desarrollo de liberación continua o rolling update ante actualizaciones por parte del software perteneciente al servicio.

2.2. Arquitectura

Para el desarrollo de la arquitectura del sistema se ha optado por utilizar una arquitectura cliente-servidor basada en 3 capas. Este estilo de despliegue arquitectónico describe la separación de la funcionalidad en segmentos separados de forma muy parecida al estilo de capas, pero en el cual cada segmento está localizado en un computador físicamente separado. Este estilo ha evolucionado desde la aproximación basada en componentes generalmente usando métodos específicos de comunicación asociados a una plataforma en vez de la aproximación basada en mensajes.

Los principios fundamentales de este tipo de arquitectura son:

- Es un estilo para definir el despliegue de las capas en una instalación.
- La arquitectura de 3 capas está caracterizada por la descomposición funcional de la aplicación, los componentes de servicio y su instalación distribuida. Mejorando la escalabilidad, disponibilidad, administración, y utilización de recursos.
- Cada capa es completamente independiente de las otras capas, excepto aquella que esta inmediatamente debajo de ella. La capa n solo necesita saber cómo manejar una solicitud de la capa n+1, como hacer la solicitud a la capa n-1 (si existe) y cómo manejar el resultado de la petición.
- La arquitectura tiene tres capas separadas o partes, cada una de ellas con su responsabilidad y está localizada en diferentes servidores.
- Una capa es desplegada en un nivel específico si más de un servicio o aplicación está expuesto por esa capa.

Inicialmente, hay que mencionar que todas las capas de las que consta el sistema se encuentran dentro del clúster de Kubernetes y los servicios que estas ofrecen para realizar las tareas demandadas por los clientes pueden ser replicados las veces necesarias para que el sistema este dotado de una alta disponibilidad como objetivo más importante.

2.2.1. Arquitectura del sistema

A continuación se muestran las capas y diferentes funcionalidades de la arquitectura, aunque más adelante se mostrarán más a fondo.

La arquitectura del sistema se compone de las siguientes capas:

- **Capa de datos**: contiene clases que interactúan con la base de datos, éstas clases altamente especializadas se encuentran en la arquitectura del sistema y permiten, utilizando los procedimientos almacenados generados, realizar todas las operaciones con la base de datos de forma transparente para la capa de negocio. Los servicios de esta capa están protegidos del acceso directo de los componentes de cliente que residen en una red segura. La interacción debe producirse a través de los procesos de la capa de negocio. Esta compuesta por la base de datos MongoDB que almacena la información correspondiente a los despliegues creados por lo docentes de la Escuela Politécnica de Cáceres para los laboratorios de las asignaturas impartidas por los mismos. Si se cuenta con mucha carga en el sistema puede replicarse esta base de datos.

- **Capa de negocio**: es donde residen los programas que se ejecutan, se reciben las peticiones del usuario y se envían las respuestas tras el proceso. En ella se establecen todas las reglas que deben cumplirse. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los resultados, y con la capa de datos, para solicitar al gestor de base de datos almacenar o recuperar datos de él. Proporciona escalabilidad al sistema, contando con uno o varios servidores en función de la carga.

- **Capa de presentación**: es la capa que ve el usuario, presenta el sistema al usuario, le comunica la información y captura la información del usuario en un mínimo de proceso. También es conocida como interfaz gráfica y debe tener la característica de ser entendible y fácil de usar para el usuario. Esta capa se comunica únicamente con la capa de negocio.

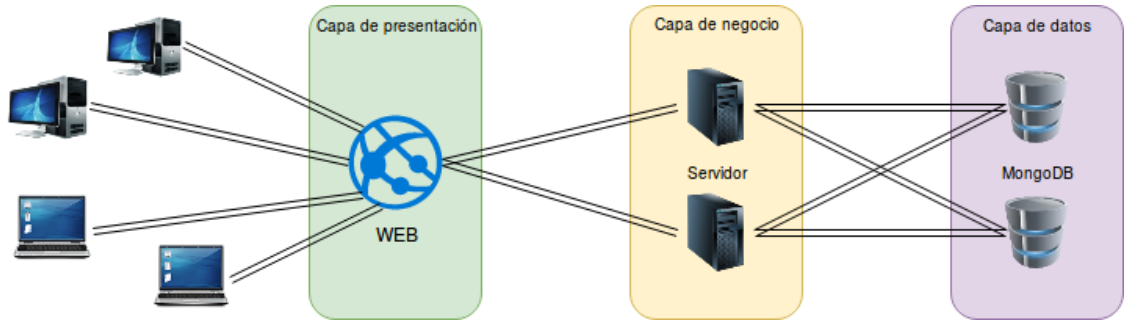


Figura 2.1: Diagrama de la arquitectura del sistema

2.3. Casos de uso

Después de establecer los requisitos para el sistema, debemos definir los casos de uso del mismo. En ellos se expondrán las principales funciones con las que cuenta para que puedan comprenderse mejor.

Lo siguientes casos de uso nacen para satisfacer las necesidades de los requisitos y de las acciones que debe cumplir el programa desarrollado.

- El usuario solicita que le sea servida la web.
- El usuario consulta el balanceo de carga de la web en el clúster.
- El usuario inserta datos en la base de datos.
- El usuario/script consulta toda la información relacionada con despliegues almacenada en la base de datos.
- El usuario/aplicación cierra la conexión.

2.3.1. El usuario solicita que le sea servida la web

El usuario de la aplicación ingresa la dirección en el navegador para que la API REST le de el código web correspondiente. Cuando recibe esta petición, la API REST proporcionará al usuario el archivo con el código web, el cual sera cargado en el navegador. Si el servicio demandado esta activo siempre se responderá a esta petición.

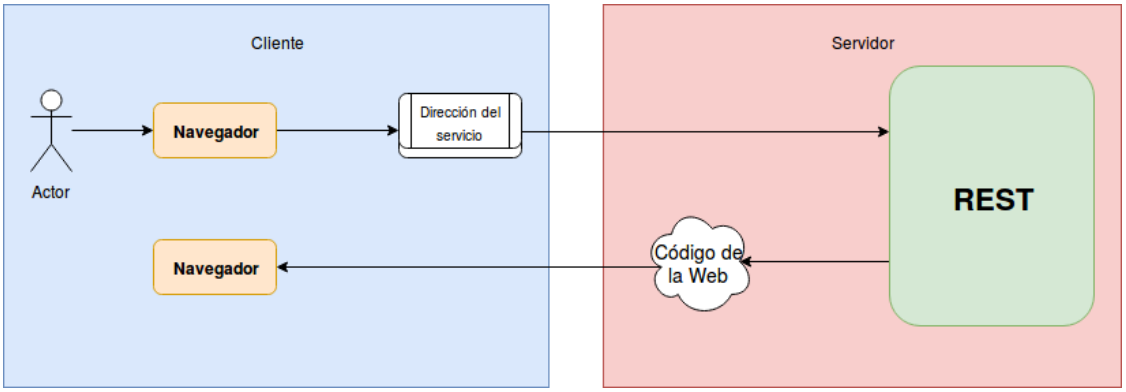


Figura 2.2: CU01 en el que el usuario solicita que le sea servida la web

2.3.2. El usuario consulta el balanceo de carga del servicio en el clúster

El usuario realiza en el navegador la selección de la pestaña relativa a test de balanceo de carga, este comando será enviado a la API REST, la cual consulta en que nodo del clúster se esta ejecutando el servicio y devuelve la respuesta al navegador.

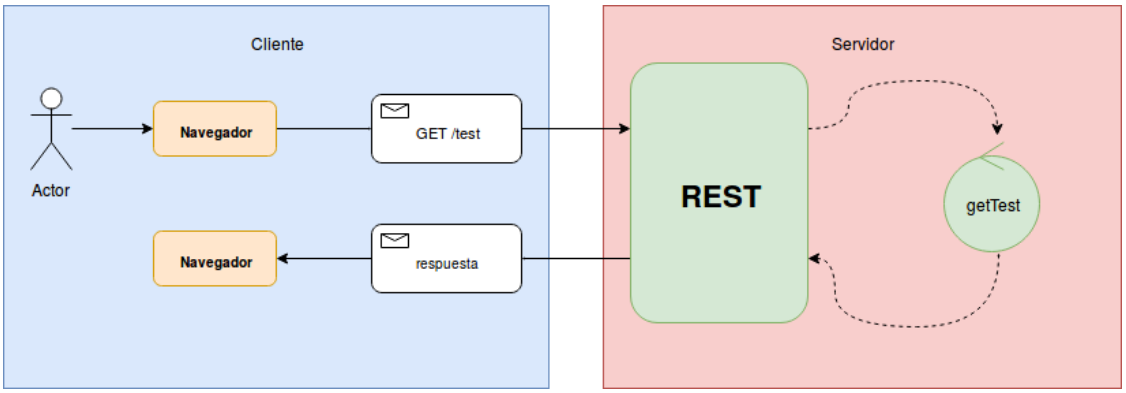


Figura 2.3: CU02 en el que el usuario consulta el balanceo de carga del servicio en el clúster.

2.3.3. El usuario inserta un despliegue en la base de datos

El usuario realiza en el navegador la selección de la pestaña relativa a la inserción de un nuevo despliegue e inserta los datos para el nuevo despliegue, este comando será enviado a la API REST, la cual realiza la inserción en la base de datos y devuelve la información relativa a todos los despliegues insertados en la misma.

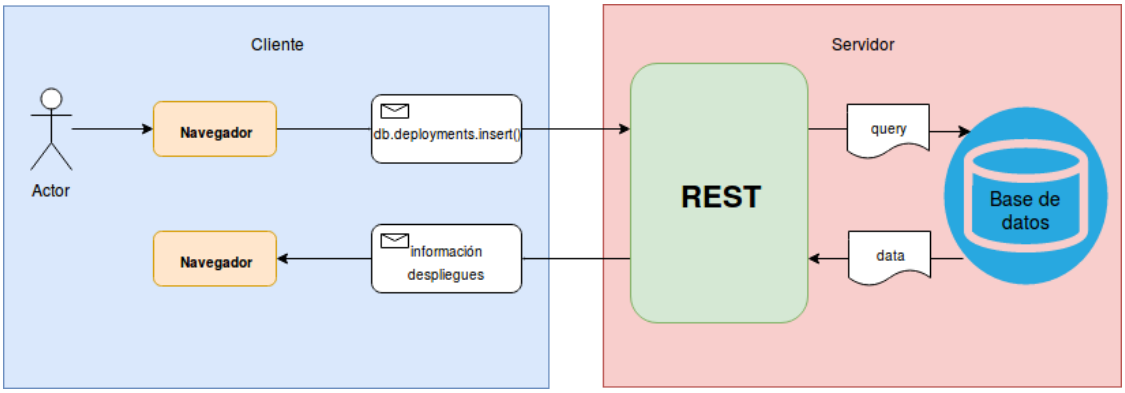


Figura 2.4: CU03 en el que el usuario inserta un despliegue en la base de datos.

2.3.4. El usuario inserta un despliegue erróneo en la base de datos

El usuario realiza en el navegador la selección de la pestaña relativa a la inserción de un nuevo despliegue e inserta los datos erróneos para el nuevo despliegue, este comando será enviado a la API REST, la cual no realiza la inserción en la base de datos y devuelve la información relativa al error producido.

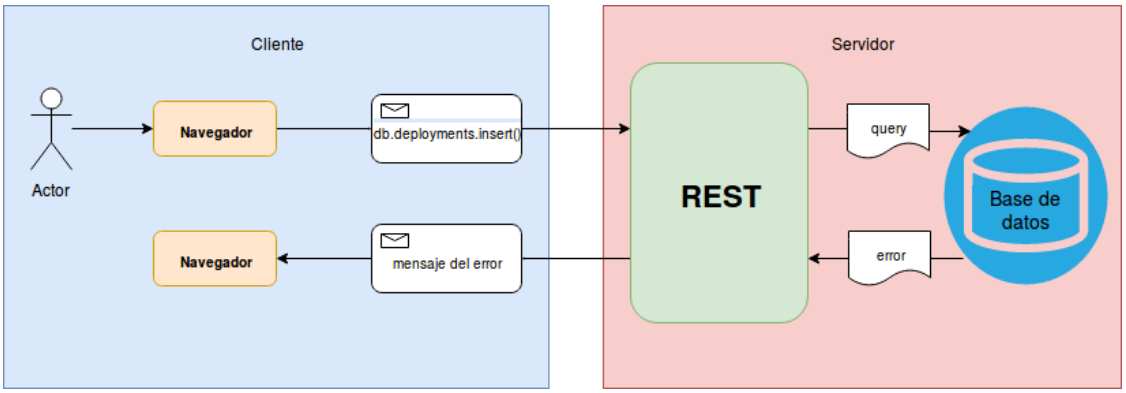


Figura 2.5: CU04 en el que el usuario inserta un despliegue erróneo en la base de datos.

2.3.5. El usuario/script actualiza un despliegue en la base de datos

El usuario realiza en el navegador la selección de la actualización de un despliegue e inserta los nuevos datos para el despliegue, este comando será enviado a la API REST, la cual realiza actualización de los datos del despliegue en la base de datos y devuelve la información relativa a todos los despliegues insertados en la misma. En el caso del script será completamente igual excepto que los comandos enviados y recibidos directamente en vez de a través de un navegador.

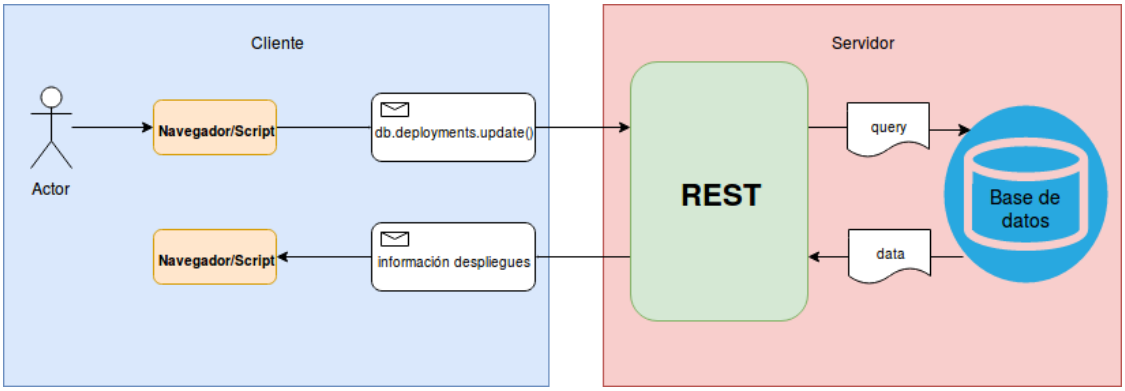


Figura 2.6: CU05 en el usuario/script actualiza un despliegue en la base de datos.

2.3.6. El usuario/script actualiza erróneamente un despliegue en la base de datos

El usuario realiza en el navegador la selección de la actualización de un despliegue e inserta los nuevos datos erróneos para el despliegue, este comando será enviado a la API REST, la cual no realiza la actualización en la base de datos y devuelve la información relativa al error producido. En el caso del script será completamente igual excepto que los comandos enviados y recibidos directamente en vez de a través de un navegador.

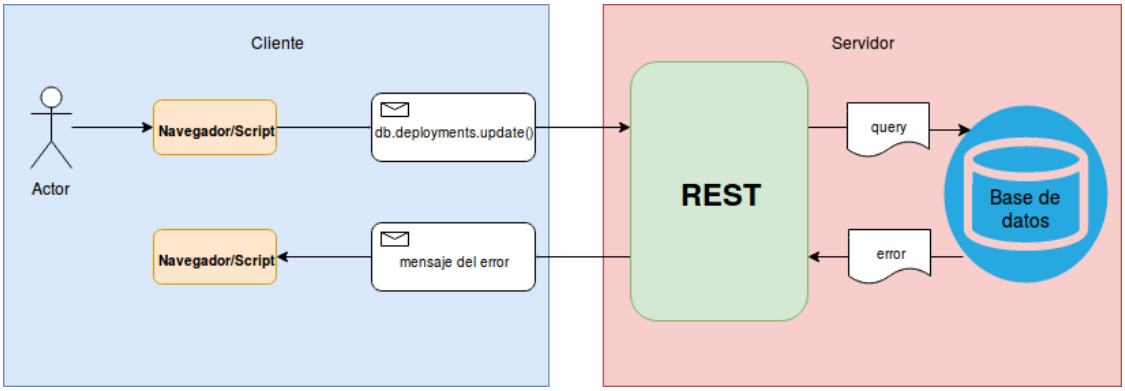


Figura 2.7: CU06 en el usuario/script actualiza erróneamente un despliegue en la base de datos.

2.3.7. El usuario/script consulta toda la información relacionada con despliegues almacenada en la base de datos

El usuario o el script que corre en un demonio para desplegar las máquinas virtuales, envía un comando y la API REST realiza la consulta para proporcionar toda la información requerida y le devuelve en un mensaje. En el caso del usuario, este comando sera enviado al realizar la selección de la pestaña relativa a la consulta de despliegues en la página web servida en el navegador.

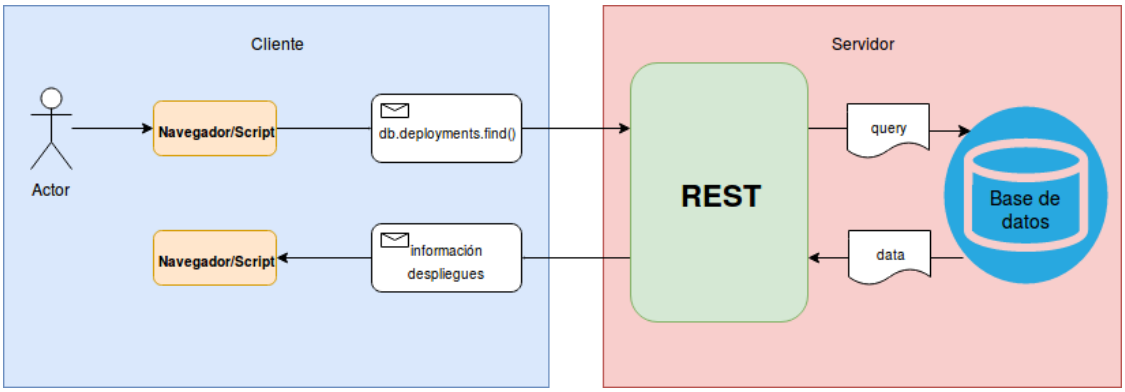


Figura 2.8: CU07 en el que el usuario/script consulta toda la información relacionada con despliegues almacenada en la base de datos.

2.3.8. El usuario/script cierra la conexión

Este caso se realiza de forma automática al cerrar la web o el script, en ese momento se envía un mensaje de cierre al servidor y este realiza la clausura de la conexión o canal establecido.

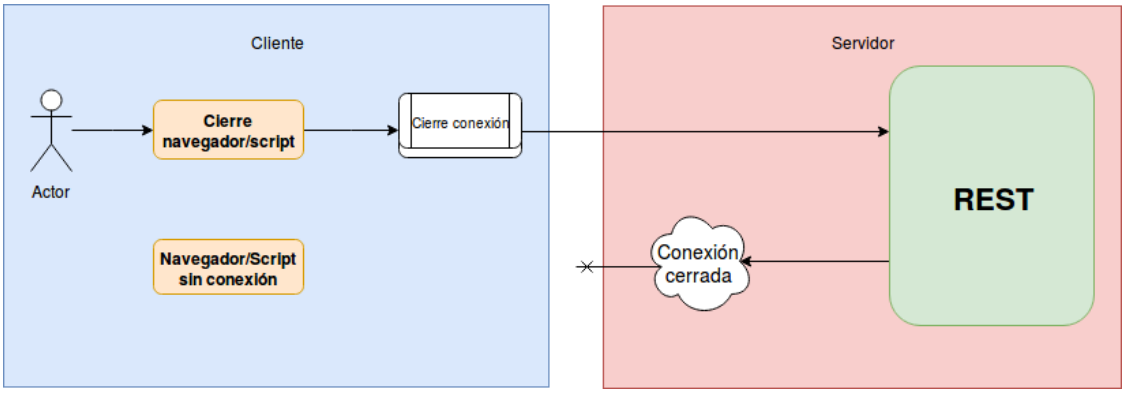


Figura 2.9: CU08 en el que el usuario/script cierra la conexión.

Capítulo 3

Tecnologías utilizadas

En esta sección se dan a conocer con más profundidad las tecnologías utilizadas para el desarrollo del sistema con la finalidad de facilitar al lector la comprensión de la implementación del producto final.

3.1. MongoDB

MongoDB es un sistema de base de datos NoSQL orientado a documentos, desarrollado bajo el concepto de código abierto.

MongoDB forma parte de la nueva familia de sistemas de base de datos NoSQL. En lugar de guardar los datos en tablas como se hace en las base de datos relacionales, MongoDB guarda estructuras de datos en documentos similares a JSON con un esquema dinámico (MongoDB utiliza una especificación llamada BSON), haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

MongoDB está escrito en C++, aunque las consultas se hacen pasando objetos JSON como parámetro. Es algo bastante lógico, dado que los propios documentos se almacenan en BSON.

MongoDB cuenta con las siguientes características principales: consultas Ad hoc, indexación, replicación, balanceo de carga, almacenamiento de archivos, agregación, ejecución de JavaScript del lado del servidor.

3.2. Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.

Python es un lenguaje de programación multiparadigma. Esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: programación orientada a objetos, programación imperativa y programación funcional. Otros paradigmas están soportados mediante el uso de extensiones.

Python usa tipado dinámico y conteo de referencias para la administración de memoria.

Una característica importante de Python es la resolución dinámica de nombres; es decir, lo que enlaza un método y un nombre de variable durante la ejecución del programa.

Otro objetivo del diseño del lenguaje es la facilidad de extensión. Se pueden escribir nuevos módulos fácilmente en C o C++. Python puede incluirse en aplicaciones que necesitan una interfaz programable.

Los usuarios de Python se refieren a menudo a la filosofía Python que es bastante análoga a la filosofía de Unix. El código que sigue los principios de Python de legibilidad y transparencia se dice que es "pythonico". Contrariamente, el código opaco u ofuscado es bautizado como "no pythonico". El desarrollador de Python Tim Peters describió los siguientes principios de esta filosofía:

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Disperso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Lo práctico gana a lo puro.
- Los errores nunca deberían dejarse pasar silenciosamente.
- A menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una -y preferiblemente sólo una- manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ya mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!

Todas estas características anteriores dotan al lenguaje de gran cantidad de ventajas: simplificado y rápido, elegante y flexible, programación sana y productiva, ordenado y limpio, portable.

```
sumaa = 0

n=1

while n < 9:
    sumaa = sumaa + n
    n = n+1

print 'La suma total es: ', sumaa
```

Figura 3.1: Ejemplo de código Python.

3.3. JavaScript

JavaScript es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

Se utiliza principalmente en su forma del lado del cliente, implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas aunque existe una forma de JavaScript del lado del servidor. Su uso en aplicaciones externas a la web es también significativo.

JavaScript se diseñó con una sintaxis similar a C, aunque adopta nombres y convenciones del lenguaje de programación Java.

Todos los navegadores modernos interpretan el código JavaScript integrado en las páginas web. Para interactuar con una página web se provee al lenguaje JavaScript de una implementación del Document Object Model.

Actualmente es ampliamente utilizado para enviar y recibir información del servidor junto con ayuda de otras tecnologías. JavaScript se interpreta en el agente de usuario al mismo tiempo que las sentencias van descargándose junto con el código HTML.

Las características principales de es lenguaje son:

- Imperativo y estructurado.
- Dinámico:

  Tipado dinámico.

  Objetual.

  Evaluación en tiempo de ejecución.
- Funcional:

  Funciones de primera clase.

- Prototípico:

  Prototipos.

  Funciones como constructores de objetos.

- Otras características:

  Entorno de ejecución.

  Funciones variádicas.

  Funciones como métodos.

  Arrays y la definición literal de objetos.

  Expresiones regulares.

3.3.1. Node.js

Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google. Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables. Fue creado por Ryan Dahl en 2009 y su evolución está apadrinada por la empresa Joyent.

Al contrario que la mayoría del código JavaScript, no se ejecuta en un navegador, sino en el servidor. Node.js implementa algunas especificaciones de CommonJS.7 Node.js incluye un entorno REPL para depuración interactiva.

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 1337;

http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}).listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Figura 3.2: Versión de un hola mundo de un servidor HTTP escrito en Node.js.

Los aspectos técnicos con los que cuenta este entorno de ejecución son:

- Concurrencia.
- Entorno de ejecución V8.
- Módulos.
- Desarrollo homogéneo entre cliente y servidor.
- Bucle de eventos.

Express

Express es el framework web más popular de Node, y es la librería subyacente para un gran número de otros frameworks web de Node populares. Proporciona mecanismos para:

- Escritura de manejadores de peticiones con diferentes verbos HTTP en diferentes caminos URL (rutas).
- Integración con motores de renderización de "vistas"para generar respuestas mediante la introducción de datos en plantillas.
- Establecer ajustes de aplicaciones web como qué puerto usar para conectar, y la localización de las plantillas que se utilizan para renderizar la respuesta.
- Añadir procesamiento de peticiones "middleware.ªdicional en cualquier punto dentro de la tubería de manejo de la petición.

Los desarrolladores han creado paquetes de middleware compatibles para abordar casi cualquier problema de desarrollo web. Hay librerías para trabajar con cookies, sesiones, inicios de sesión de usuario, parámetros URL, datos POST, cabeceras de seguridad y muchos más.

Entre los diferentes middlewares con los que cuenta Express, se ha utilizado **express-handlebars** que proporciona la potencia necesaria para permitirte construir plantillas semánticas de manera efectiva. Permiten precompilar las plantillas e incluirlas como código JavaScript.

Handlebars agrega un par de características adicionales para facilitar la escritura de plantillas y también cambia un pequeño detalle de cómo funcionan los parciales:

- Caminos anidados haciendo posible buscar propiedades anidadas debajo del contexto actual.
- Se puede acceder a los ayudantes de handlebars desde cualquier contexto en una plantilla.

- Las expresiones de bloque permiten definir ayudantes que invocarán una sección de su plantilla con un contexto diferente al actual. Estos ayudantes de bloque se identifican por un # que precede al nombre del ayudante y requieren un bigote de cierre correspondiente, /, del mismo nombre.
- Las llamadas de ayuda también pueden tener valores literales pasados como argumentos de parámetros o argumentos hash.
- Puede usar los comentarios en el código de handlebars de manera similar a un comentario en código.

Method-override

El middleware Method-override es para solicitudes de clientes que solo admiten verbos simples como GET y POST. Entonces, en esos casos, podría especificar un campo de consulta especial que indique el verbo real a usar en lugar de lo que se envió originalmente. Las rutas no tienen que cambiar y seguirán funcionando y puede aceptar solicitudes de todo tipo de clientes.

Mongoose

Mongoose es una biblioteca de JavaScript que permite definir esquemas con datos fuertemente tipados. Una vez que se define un esquema, Mongoose permite crear un modelo basado en un esquema específico. Un modelo de Mongoose se asigna a un documento MongoDB a través de la definición del esquema del modelo.

Mongoose es un Object Document Mapper (ODM), Esto significa que permite definir objetos con un esquema fuertemente tipado que se asigna a un documento MongoDB.

Esta biblioteca proporciona una increíble cantidad de funcionalidades para crear y trabajar con esquemas. Actualmente contiene ocho SchemaTypes: string, number, date, buffer, boolean, mixed, objectId y array.

Cada tipo de datos le permite especificar:

- Un valor predeterminado.
- Una función de validación personalizada.
- Indica que se requiere un campo.
- Una función get que le permite manipular los datos antes de que se devuelva como un objeto.
- Una función de conjunto que le permite manipular los datos antes de guardarlos en la base de datos.
- Crear índices para permitir que los datos se obtengan más rápido.

Además de estas opciones comunes, ciertos tipos de datos le permiten personalizar aún más cómo se almacenan y recuperan los datos de la base de datos.

Connect-flash

El flash es un área especial de la sesión utilizada para almacenar mensajes. Los mensajes se escriben en el flash y se borran después de mostrarse al usuario. El flash se suele utilizar en combinación con redirecciones, lo que garantiza que el mensaje esté disponible en la página siguiente que se va a representar.

Nodemon

Nodemon es un monitor que facilita el desarrollo de una aplicación basada en node.js. Su función principal es la de vigilar los archivos del directorio en el que iniciemos nodemon. Si estos cambian, Nodemon reiniciará automáticamente la aplicación.

Nodemon no requiere cambios en el código o en el método de desarrollo, simplemente envuelve la aplicación nodejs y mantiene un ojo en los archivos que han cambiado.

Las características de este monitor son las siguientes:

- Reinicio automático de la aplicación.
- Detecta la extensión de archivo por defecto para monitorear.
- Soporte predeterminado para node y coffeescript, pero fácil de ejecutar cualquier ejecutable.
- Ignorando archivos o directorios específicos.
- Ver directorios específicos.
- Funciona con aplicaciones de servidor o con utilidades de ejecución de una sola vez y REPLs(Bucle de Lectura-Evaluación-Impresión).
- Requerido en aplicaciones de node.
- Fuente abierta y disponible en github.

3.3.2. jQuery

jQuery es una biblioteca multiplataforma de JavaScript, que permite simplificar la manera de interactuar con los documentos HTML, manipular el árbol DOM, manejar eventos, desarrollar animaciones y agregar interacción con la técnica AJAX a páginas web.

jQuery es software libre y de código abierto, ofrece una serie de funcionalidades basadas en JavaScript que de otra manera requerirían de mucho más código.

Las características de esta biblioteca son las siguientes:

- Selección de elementos DOM.
- Interactividad y modificaciones del árbol DOM, incluyendo soporte para CSS 1-3 y un plugin básico de XPath.
- Eventos.
- Manipulación de la hoja de estilos CSS.
- Efectos y animaciones.
- Animaciones personalizadas.
- AJAX.
- Soporta extensiones.
- Utilidades varias como obtener información del navegador, operar con objetos y vectores, funciones para rutinas comunes, etc.
- Compatible con los navegadores Mozilla Firefox 2.0+, Internet Explorer 6+, Safari 3+, Opera 10.6+ y Google Chrome 8+.

3.3.3. JSON

JSON es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript. JSON es un formato de texto que es completamente independiente del lenguaje pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

JSON está constituído por dos estructuras:

- Una colección de pares de nombre/valor.
- Una lista ordenada de valores.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras. Los tipos de datos disponibles con JSON son números, cadenas, booleanos, null, arrays y objetos.

```
{ "pedido":  {
   "fecha": "2017-05-29",
   "productos": [
    { "codigo" : "R23" ,
      "nombre" : "Rotulador",
      "cantidad" : 20
    },
    { "codigo" : "G56" ,
      "nombre" : "Grapadora",
      "cantidad" : 2
     }
  ]
 }
}
```

Figura 3.3: Ejemplo de código JSON.

3.4. REST

La transferencia de estado representacional o REST es un estilo de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web. Describe cualquier interfaz entre sistemas que utilice directamente HTTP para obtener datos o indicar la ejecución de operaciones sobre los datos, en cualquier formato (XML, JSON, etc) sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes. Es posible diseñar sistemas de servicios web de acuerdo con el estilo arquitectural REST de Fielding y también es posible diseñar interfaces XMLHTTP de acuerdo con el estilo de llamada a procedimiento remoto.

Cuenta con una serie de características fundamentales clave:

- Protocolo cliente/servidor sin estado.
- Las operaciones más importantes relacionadas con los datos en cualquier sistema REST: POST(crear), GET(leer y consultar), PUT(editar) y DELETE(eliminar).
- Los objetos en REST siempre se manipulan a partir de la URI.
- Interfaz uniforme.
- Sistema de capas.
- Uso de hipermedios.
- Principio HATEOAS (Hypermedia As The Engine Of Application State - Hipermedia Como Motor del Estado de la Aplicación)

A la hora de realizar un desarrollo, ofrece las siguientes ventajas:

- Separación entre el cliente y el servidor.
- Visibilidad, fiabilidad y escalabilidad.
- La API REST siempre es independiente del tipo de plataformas o lenguajes.

Figura 3.4: Diagrama simple de una API REST.

3.5. HTML

HTML es un lenguaje de marcado que se utiliza para el desarrollo de páginas de Internet. Se trata de la sigla que corresponde a HyperText Markup Language, es decir, Lenguaje de Marcas de Hipertexto, que podría ser traducido como Lenguaje de Formato de Documentos para Hipertexto.

Se trata de un formato abierto que surgió a partir de las etiquetas SGML (Standard Generalized Markup Language). Concepto traducido generalmente como "Estándar de Lenguaje de Marcado Generalizado" y que se entiende como un sistema que permite ordenar y etiquetar diversos documentos dentro de una lista. Este lenguaje es el que se utiliza para especificar los nombres de las etiquetas que se utilizarán al ordenar, no existen reglas para dicha organización, por eso se dice que es un sistema de formato abierto.

EL HTML se encarga de desarrollar una descripción sobre los contenidos que aparecen como textos y sobre su estructura, complementando dicho texto con diversos objetos.

Es un lenguaje muy simple y general que sirve para definir otros lenguajes que tienen que ver con el formato de los documentos. El texto en él se crea a partir de etiquetas, también llamadas tags, que permiten interconectar diversos conceptos y formatos.

El HTML se escribe en forma de «etiquetas», rodeadas por corchetes angulares (<,>,/). El HTML también puede describir, hasta un cierto punto, la apariencia de un documento. También sirve para referirse al contenido del tipo de MIME text/html o todavía más ampliamente como un término genérico para el HTML, ya sea en forma descendida del XML o en forma descendida directamente de SGML.

HTML consta de varios componentes vitales, entre ellos los **elementos** y sus **atributos**, tipos de datos y la declaración de tipo de documento.

Por otra parte, cabe destacar que el HTML permite ciertos códigos que se conocen como scripts, los cuales brindan instrucciones específicas a los navegadores que se encargan de procesar el lenguaje. Entre los scripts que pueden agregarse, los más conocidos y utilizados son JavaScript y PHP.

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Example<title>
5          <link rel="stylesheet" href="styl
6      </head>
7      <body>
8          <h1>
9              <a href="/">Header</a>
10         </h1>
11         <nav>
12             <a href="one/">One</a>
13             <a href="two/">Two</a>
14             <a href="three/">Three</a>
15         </nav>
```

Figura 3.5: Ejemplo de código HTML.

3.6. CSS

CSS, Hojas de estilo en cascada, es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado. Es muy usado para establecer el diseño visual de los documentos web, e interfaces de usuario escritas en HTML o XHTML; el lenguaje puede ser aplicado a cualquier documento XML. También permite aplicar estilos no visuales.

Está diseñado principalmente para marcar la separación del contenido del documento y la forma de presentación de este, características tales como las capas o layouts, los colores y las fuentes. Esta separación busca mejorar la accesibilidad del documento, proveer más flexibilidad y control en la especificación de características presentacionales, permitir que varios documentos HTML compartan un mismo estilo usando una sola hoja de estilos separada en un archivo .css, y reducir la complejidad y la repetición de código en la estructura del documento.

Una hoja de estilos consiste en una serie de reglas. Cada regla, o conjunto de reglas consisten en uno o más **selectores**, y un **bloque de declaración**.

Algunas de las ventajas al utilizar CSS son: separación del contenido y la presentación, consistencia del sitio.ancho de banda, formateo de página y accesibilidad.

```
1  h1 {color:green;
2  background-color:yellow;
3  font-size:40px;}
4
5  h2 {color:red;
6  background-color:blue;
7  font-size:30px}
8
9  p {color:black; font-size:25px}
```

Figura 3.6: Ejemplo de código CSS.

3.7. Bootstrap

Bootstrap es una biblioteca multiplataforma o conjunto de herramientas de código abierto para diseño de sitios y aplicaciones web. Contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basado en HTML y CSS, así como extensiones de JavaScript adicionales. A diferencia de muchos frameworks web, solo se ocupa del desarrollo front-end.

Bootstrap es modular y consiste esencialmente en una serie de hojas de estilo LESS que implementan la variedad de componentes de la herramienta. Una hoja de estilo llamada bootstrap.less incluye los componentes de las hojas de estilo. Los desarrolladores pueden adaptar el mismo archivo de Bootstrap, seleccionando los componentes que deseen usar en su proyecto.

Los ajustes son posibles en una medida limitada a través de una hoja de estilo de configuración central. Los cambios más profundos son posibles mediante las declaraciones LESS.

El uso del lenguaje de hojas de estilo LESS permite el uso de variables, funciones y operadores, selectores anidados, así como clases mixin.

Bootstrap cuenta con las siguientes características:

- Sistema de cuadrilla y diseño sensible.
- Entendiendo la hoja de estilo CSS.
- Componentes re-utilizables.
- Plug-ins de JavaScript.

Entre las ventajas que proporciona el uso de Bootstrap se encuentran: simplificar el proceso de maquetación, web bien organizada de forma visual rápidamente, utilizar muchos elementos web, diseño adaptable, grid system, buena integración con las principales librerías Javascript y implementaciones externas para WordPress, Drupal, etc.

3.8. Docker

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos. Docker utiliza características de aislamiento de recursos del kernel Linux para permitir que contenedores independientes se ejecuten dentro de una sola instancia de Linux, evitando la sobrecarga de iniciar y mantener máquinas virtuales.

Incluye la biblioteca libcontainer como su propia manera de utilizar directamente las facilidades de virtualización que ofrece el kernel Linux, además de utilizar las interfaces abstraídas de virtualización mediante libvirt, LXC y systemd-nspawn.

Es una herramienta que puede empaquetar una aplicación y sus dependencias en un contenedor virtual que se puede ejecutar en cualquier servidor Linux. Esto ayuda a permitir la flexibilidad y portabilidad en donde la aplicación se puede ejecutar, ya sea en las instalaciones físicas, la nube pública, nube privada, etc.

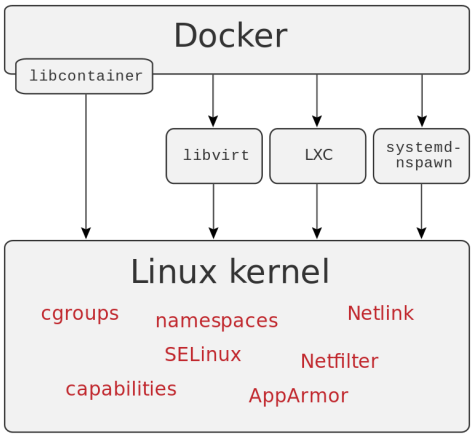


Figura 3.7: Interfaces utilizadas por Docker para acceder a las capacidades de virtualizacion del kernel Linux.

Para comprender Docker en su totalidad, hay que entender dos conceptos básicos en Docker: las imágenes y los contenedores.

3.8.1. Imágenes

Una imagen de Docker, es una estructura de directorios y paquetes mínima, creada para una función básica. Se trata de que sea una plantilla, que se puede modificar, pero que ha sido pensada para una uso básico y específico.

Cada imagen está compuesta por una serie de capas donde se almacenan los ficheros. Esta división en capas incrementa la reusabilidad, disminuye el uso de disco y acelera el proceso de construcción de la imagen. Una vez se construye una imagen, se puede ejecutar dicho contenedor usando el comando básico de Docker, docker run. Cuando se lanza un contenedor lo que se hace es lanzar esa imagen creada y se añade una nueva capa a la misma: la capa de contenedor.

Dockerfiles

Como ya se ha explicado en el apartado anterior, una imagen de Docker está formada por todos los ficheros que necesita la aplicación que se va a lanzar en el contenedor. Para construir esta imagen, Docker hace una división en capas, donde cada una de estas capas consiste en una serie de instrucciones Estas instrucciones realizadas son guardados en un fichero llamado Dockerfile.

```
# A basic apache server
FROM ubuntu:14.04

MAINTAINER Chuck Norris: 0.1

RUN apt-get update
RUN apt-get install -y apache2
RUN apt-get clean

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2

EXPOSE 80

CMD ["/usr/sbin/apache2", "-D",
"FOREGROUND"]
```

Figura 3.8: Ejemplo de Dockerfile.

A partir del Dockerfile de la imagen, es posible construir una imagen con el comando docker build. Una vez construida la imagen, el siguiente paso es iniciar el contenedor.

3.8.2. Contenedores

Un contenedor, es una instancia generada desde una imagen, que ya es en sí un pequeño OS funcionando, y que podemos usarlo para una función concreta.

Mediante el uso de contenedores, los recursos pueden ser aislados, los servicios restringidos, y se otorga a los procesos la capacidad de tener una visión casi completamente privada del sistema operativo con su propio identificador de espacio de proceso, la estructura del sistema de archivos, y las interfaces de red.

Usando Docker para crear y gestionar contenedores puede simplificar la creación de sistemas altamente distribuidos. Permite que el despliegue de nodos se realice a medida que se dispone de recursos o cuando se necesiten más nodos, lo que permite una plataforma como servicio (PaaS - Plataform as a Service) de estilo de despliegue y ampliación de los sistemas como Apache Cassandra, MongoDB o Riak. Docker también simplifica la creación y el funcionamiento de las tareas de carga de trabajo o las colas y otros sistemas distribuidos.

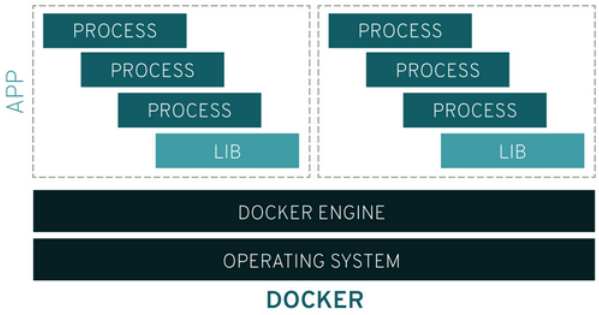


Figura 3.9: Contenedor Docker.

El uso de estos contenedores dotan al sistema de una serie de ventajas: modularidad, capas y control de versión de imagen, restauración y implementación rápida.

3.9. Libvirt

Libvirt es una biblioteca de virtualización que brinda una API agnóstica a hipervisores para administrar de manera segura sistemas operativos huéspedes que se ejecutan en un host. Es una API para construir herramientas a fin de administrar sistemas operativos huéspedes. Ofrece una API común para la funcionalidad común que implementan los hipervisores soportados.

Libvirt existe como un conjunto de interfaces de programación de aplicaciones diseñado para ser utilizado por una aplicación de gestión. A través de un mecanismo específico en hipervisores, se comunica con cada hipervisor disponible para realizar solicitudes de API.

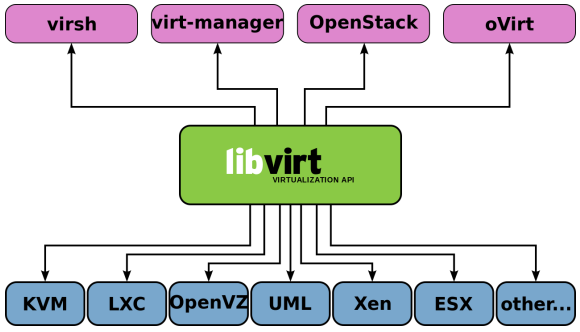


Figura 3.10: Tecnologías soportadas por libvirt.

3.9.1. QEMU

QEMU es un emulador de procesadores basado en la traducción dinámica de binarios. Esta máquina virtual puede ejecutarse en cualquier tipo de microprocesador o arquitectura.

QEMU es capaz de emular una máquina completa en software sin ninguna necesidad de soporte de virtualización de hardware. Con soporte de hipervisor, QEMU puede lograr un rendimiento cercano al nativo para CPUs. También es capaz de proporcionar virtualización de API de espacio de usuario para Linux y las interfaces del núcleo BSD.

QEMU tiene como objetivo adaptarse a una variedad de casos de uso. Se puede invocar directamente por los usuarios que desean tener un control total sobre su comportamiento y configuración. También tiene como objetivo facilitar la integración en la gestión de capas de nivel superior.

Posee dos modos de operación:

- Emulación del modo usuario.
- Modo de emulación completo de sistema de ordenador.

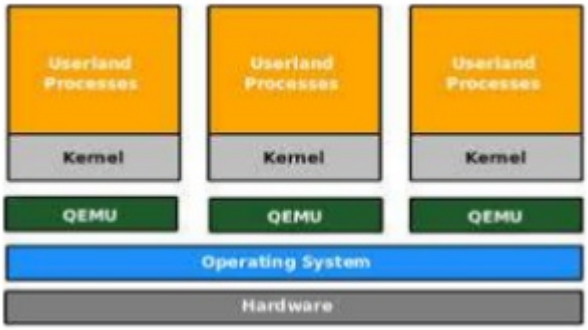


Figura 3.11: QEMU funcionando de manera independiente.

3.9.2. KVM

La máquina virtual basada en el kernel o KVM es una tecnología de virtualización de open source integrada a Linux. En concreto, con KVM puede convertir a Linux en un hipervisor que permite que una máquina de host ejecute entornos virtuales múltiples y aislados llamados máquinas virtuales (VM) o huéspedes. Está formada por un módulo del núcleo y herramientas en el espacio de usuario, siendo en su totalidad software libre. El componente KVM para el núcleo está incluido en Linux.

Permite ejecutar máquinas virtuales utilizando imágenes de disco que contienen sistemas operativos sin modificar.

Una de las características que KVM posee es el «overcommit», que es el uso de memoria excediendo aún la memoria física del host. Entre el resto de características de esta tecnología también se consideran de gran importancia: seguridad, almacenamiento, compatibilidad con el hardware, gestión de memoria, migración en vivo, rendimiento y escalabilidad, programación y control de recursos, latencia más baja y mayor priorización.

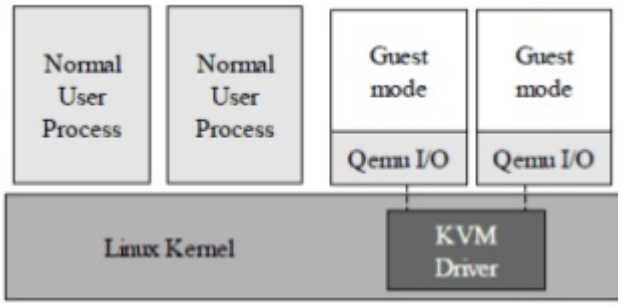


Figura 3.12: KVM funcionando junto a QEMU.

3.9.3. Virsh

Virsh es una potente herramienta de línea de comandos que nos permite gestionar múltiples aspectos de la gestión de máquinas virtuales KVM. Es la interfaz principal para administrar los dominios invitados virsh. El programa se puede utilizar para crear, pausar y cerrar dominios. También se puede utilizar para listar los dominios actuales.

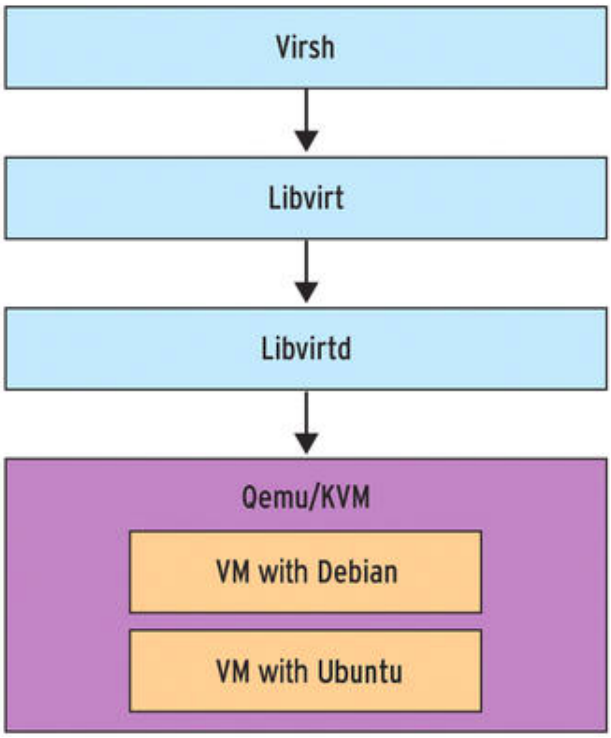


Figura 3.13: Virsh pasa los comandos a los otros componentes del paquete libvirt.

3.9.4. Virt-manager

Virt-manager es una interfaz de usuario de escritorio para administrar máquinas virtuales a través de libvirt. Se dirige principalmente a las máquinas virtuales KVM. Presenta una vista de resumen de los dominios en ejecución, su rendimiento en vivo y las estadísticas de utilización de recursos. Los asistentes permiten la creación de nuevos dominios y la configuración y ajuste de la asignación de recursos y el hardware virtual de un dominio. Un visor de cliente VNC y SPICE integrado presenta una consola gráfica completa para el dominio invitado.

La parte frontal de la aplicación utiliza las bibliotecas GTK / Glade para todos los componentes de interacción del usuario. El back-end utiliza libvirt para administrar máquinas virtuales Qemu / KVM y Xen, así como contenedores LXC. La interfaz de usuario se prueba principalmente con KVM, pero está pensada para ser razonablemente portátil a cualquier soporte de virtualización que admita libvirt.

3.10. Kubernetes

Kubernetes es un proyecto opensource de Google, diseñado para automatizar el despliegue, escalado y poder operar con aplicaciones contenerizadas. En definitiva es un orquestador de contenedores.

Kubernetes es: portable, extensible y self-healing.

Nos proporciona todo lo necesario para mantener en producción nuestros framework como: montaje de volumenes para su persistencia, distribución de secretos y gestor de la configuración, gestión de la vida del contenedor, replicación de contenedores, uso de autoescalado horizontal, descubrimiento de servicios y balanceo del framework, monitorización, acceso a los logs y debug de los framework.

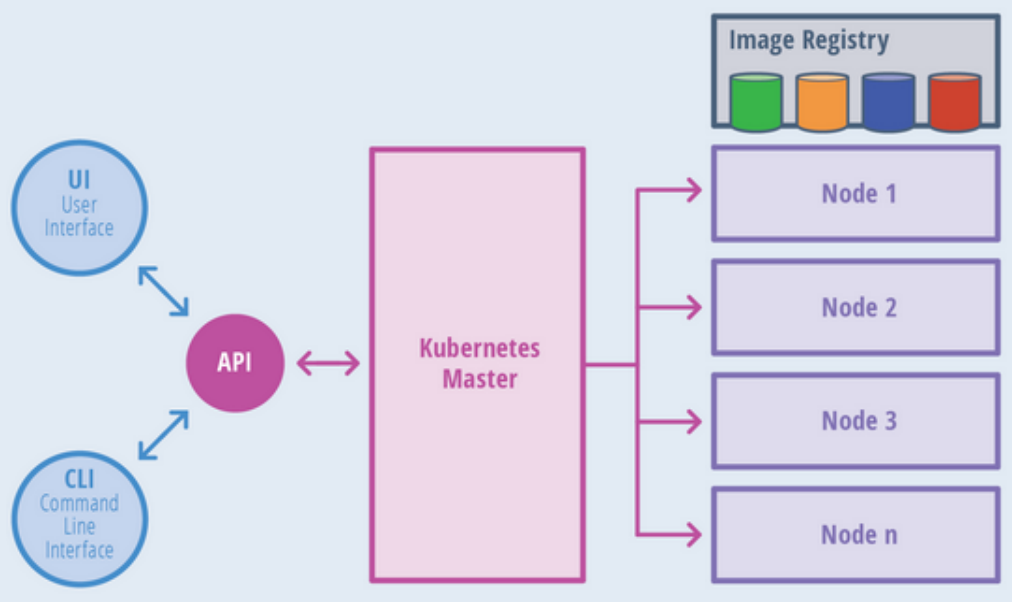


Figura 3.14: Arquitectura de Kubernetes a alto nivel.

Los conceptos básicos de esta tecnología son los siguientes:

- **Cluster:** conjunto de máquinas físicas o virtuales y otros recursos utilizados por kubernetes.
- **Nodo:** una máquina física o virtual ejecutándose en kubernetes donde pods pueden ser programados.
- **Pod:** conjunto de contenedores y volumenes. Son la unidad más pequeña desplegable.
- **Despliegue o Conjunto de réplicas, (Controladores):** gestor de pods que asegura que están levantadas las réplicas y permite escalar de forma fácil.
- **Servicios:** define como acceder a un grupo de pods.
- **Espacios de nombres:** establece un nivel adicional de separación entre los contenedores que comparten los recursos de un cluster.
- **Mapa de configuración:** servicio para gestionar la configuración de nuestras aplicaciones.
- **Secretos:** servicio para gestionar los secretos de nuestras aplicaciones.
- **Volúmenes:** servicio para gestionar la persistencia de los contenedores.

3.10.1. Componentes del cluster

Componentes del master

- **kube-apiserver:** expone la API de Kubernetes.
- **etcd:** se usa como respaldo de Kubernetes, todos los datos del cluster se almacenan aquí, manteniéndolos en alta disponibilidad.
- **kube-controller-manager:** ejecuta los controladores, se encarga de las rutinas del cluster.
- **cloud-controller-manager:** se encarga de interactuar con los proveedores de la nube.
- **kube-scheduler:** se encarga de mirar los pods creados recientemente y si no tienen asignado un nodo este los asigna.
- **addons:** son pods y servicios que implementan funcionalidades al cluster.
- **DNS:** cluster DNS es un servidor de DNS, además sirve de registro DNS para los servicios de Kubernetes.
- **Web UI (Dashboard):** permite administrar y solucionar problemas en las aplicaciones que se ejecutan en el cluster, así como en el propio cluster.
- **Monitoreo de recursos de contenedores:** graba una serie de métricas sobre los contenedores y las facilita por el WEB UI.
- **Cluster-level Logging:** es el responsable de salvar los logs de los contenedores a un registro central para posteriormente poderse buscar desde una interfaz web.

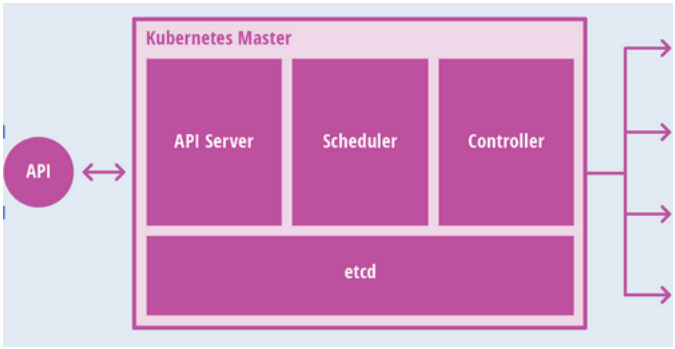


Figura 3.15: Componentes del master de Kubernetes.

Componentes de los nodos

Mantiene corriendo los pods y proporciona el entorno de tiempo de ejecución de Kubernetes. Forman parte de estos: kubelet, kube-proxy, docker, rkt, supervisord, fluentd.

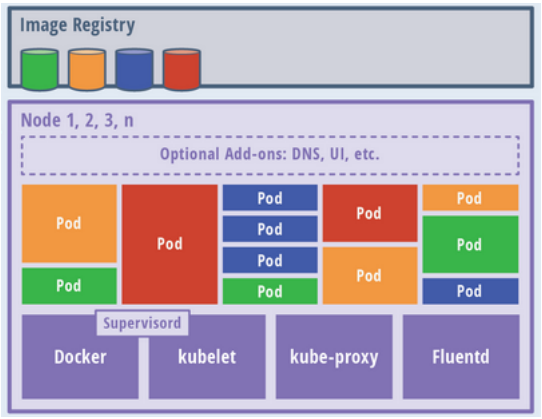


Figura 3.16: Componentes de los nodos de Kubernetes.

3.10.2. POD's

Pod es un grupo de uno o más contenedores.

Varios contenedores que pertenezcan al mismo pod son visibles unos de otros vía localhost. Los contenedores que se encuentran en distintos pods no pueden comunicarse de esta manera.

Hay que tener en cuenta que los pods son entidades efímeras. En el ciclo de vida de un pod estos se crean y se destruyen, si un nodo que contiene un pod es eliminado, todos los pods que contenía ese nodo se pierden y serán remplazados en otro nodo. Esto es importante porque un pod no debería de tener información almacenada que pueda ser utilizada después por otro pod en caso de que a este le pasara algo. Para compartir información entre pods están los volúmenes de persistencia.

La comunicación entre pods se realizara con Flannel, el cual va a generar en cada uno de los nodos una subred y luego configurará docker para que forme parte de esa subred, para esto Flannel consulta a ectd cual es la configuración de la red del cluster, con esto ya puede entender a que nodos tiene que enviar las peticiciones.

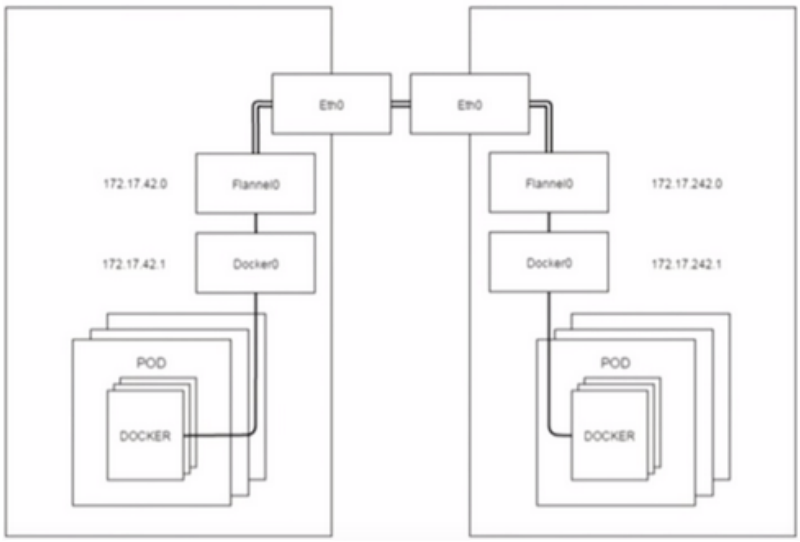


Figura 3.17: Conectividad entre pods.

3.10.3. Controladores

Aunque el despliegue es uno de los controladores mas importantes de Kubernetes, todos los tipos de controladores son:

- **Controlador de replicación:** garantiza que un número específico de réplicas de pod se estén ejecutando en cualquier momento. En otras palabras, un controlador de replicación se asegura de que un pod o un conjunto homogéneo de pods siempre esté activo y disponible.

  Si hay demasiados pods, el controlador de replicación termina los pods adicionales. Si hay muy pocos, el controlador de replicación inicia más pods. A diferencia de los pods creados manualmente, los pods mantenidos por un controlador de replicación se reemplazan automáticamente si fallan, se eliminan o se terminan. Un controlador de replicación es similar a un supervisor de procesos, pero en lugar de supervisar procesos individuales en un solo nodo, el controlador de replicación supervisa múltiples pods a través de múltiples nodos.

- **Conjuntos de réplicas:** es el controlador de replicación de próxima generación. La única diferencia entre un conjunto de réplicas y un controlador de replicación en este momento es el soporte del selector. El conjunto de réplicas admite los nuevos requisitos del selector basado en conjuntos como se describe en la guía del usuario de las etiquetas, mientras que un controlador de replicación solo admite los requisitos del selector basado en la igualdad.

- **Despliegues:** se asegura de que grupo de uno o más pods esté siempre disponible. Siempre que queramos desplegar un contenedor es muy recomendable usarlo para gestionar la vida del contenedor. Un despliegue es al fin y al cabo un supervisor de un grupo de uno o más pods a través de un conjunto de nodos. La diferencia con un conjunto de réplicas es que a la hora de actualizar nuestra aplicación lo realiza de una forma mas eficiente, si tenemos una aplicacion corriendo en 3 pod's y la actualizamos se haría de una forma ordenada para no perder servicio.

  Proporciona actualizaciones declarativas para pods y conjuntos de réplicas.

Describe un estado deseado en un objeto despliegue, y el controlador de despliegues cambia el estado real al estado deseado a una velocidad controlada. Se pueden definir despliegues para crear nuevos conjuntos de réplicas o eliminar despliegues existentes y adoptar todos sus recursos con los nuevos despliegues.

- **Conjunto con estado:** es el objeto de la API de carga de trabajo utilizado para administrar aplicaciones con estado.

  Administra el despliegue y el escalado de un conjunto de pods y proporciona garantías sobre el orden y la singularidad de estos pods.

  Al igual que en un despliegue, un conjunto con estado administra pods que se basan en una especificación de contenedor idéntica. A diferencia de un despliegue, un conjunto con estado mantiene una identidad fija para cada uno de sus pods. Estos pods se crean a partir de la misma especificación, pero no son intercambiables: cada uno tiene un identificador persistente que mantiene a través de cualquier reprogramación.

- **Conjunto de demonios:** garantiza que todos los nodos (o algunos) ejecuten una copia de un pod. A medida que se agregan nodos al clúster, se agregan pods a ellos. A medida que los nodos se eliminan del clúster, esos pods se recolectan como basura. Eliminar un conjunto de demonios limpiará los pods que creó.

- **Recolección de basura:** su función es eliminar ciertos objetos que alguna vez tuvieron un propietario, pero ya no lo tienen.

- **Trabajos:** crea uno o más pods y asegura que un número específico de ellos termine con éxito. A medida que los pods se completan con éxito, este los rastrea. Cuando se alcanza un número específico de terminaciones exitosas, el trabajo en sí está completo. Al eliminarlo limpiará los pods que creó.

  También se puede utilizar para ejecutar varios pods en paralelo.

- **Trabajos cron:** crea trabajos en un horario basado en el tiempo. Es como una línea de un archivo crontab. Ejecuta un trabajo periódicamente en un horario determinado, escrito en formato Cron.

3.10.4. Servicios

Es una abstracción que define un grupo lógico de pods y una política de acceso a los mismos. Los pods apuntan a un servicio normalmente por la propiedad label. El servicio hace que un pod siempre sea accesible de la misma manera, de forma que aunque el pod se destruya o se modifique siempre sea accesible por la abstracción.

Hay tres tipos de servicios para acceder a nuestra aplicación:

- **ClusterIP:** usa únicamente IP interna del cluster. Podremos acceder a través de una VPN o desde dentro del cluster.

- **Puerto de nodo:** todos los nodos del cluster van a exponer el puerto del servicio a través del servicio kube-proxy que también se encuentra en todos los nodos. Además de tener una IP interna en el cluster, expone el servicio en un puerto en cada nodo del cluster, el mismo puerto en cada nodo. Podrá ponerse en contacto con el servicio en cualquier dirección del puerto de nodo.

- **Balanceador de carga:** además de contar con una IP interna de cluster y un servicio de exposición en un puerto de nodo, solicita al proveedor de la nube un balanceador de carga que reenvía al servicio expuesto como puerto de nodo para cada nodo. Solo funciona para el proveedor de la nube que admite balanceadores de carga externos.

La exposición de estos servicios al exterior puede ser complementada con el uso del ingreso.

Ingreso

Ingreso es un objeto de API que administra el acceso externo a los servicios en un clúster, generalmente HTTP. Puede proporcionar balanceo de carga, terminación SSL y alojamiento virtual basado en nombre.

Expone las rutas HTTP y HTTPS desde fuera del cluster a los servicios dentro del cluster. El enrutamiento del tráfico está controlado por las reglas definidas en el recurso de ingreso.

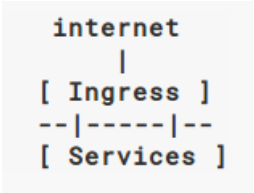


Figura 3.18: Capa de ingreso.

Puede configurarse para que brinde servicios a direcciones URL accesibles, balanceo de carga del tráfico, terminación SSL y ofrecer alojamiento virtual basado en nombre. Un controlador de ingreso es responsable de cumplir con el ingreso, generalmente con un equilibrador de carga, aunque también puede configurar su enrutador de borde o frontends adicionales para ayudar a manejar el tráfico.

Un ingreso no expone puertos o protocolos arbitrarios. La exposición de servicios distintos de HTTP y HTTPS a Internet generalmente utiliza un servicio de tipo puerto de nodo o balanceador de carga.

3.10.5. Espacios de nombres

Permiten establecer un nivel adicional de separación entre los contenedores que comparten los recursos de un cluster.

Esto es especialmente útil cuando diferentes grupos usan el mismo cluster y existe el riesgo potencial de colisión de nombres de los pods, usados por los diferentes equipos.

Los espacios de nombres también facilitan la creación de cuotas para limitar los recursos disponibles para cada espacio de nombres. Pueden considerarse los espacios de nombres como clusters virtuales sobre el cluster físico de Kubernetes. De esta forma, proporcionan separación lógica entre los entornos de diferentes equipos.

Kubernetes proporciona dos espacios de nombres por defecto: kube-system y default. A grosso modo, los objetos "de usuario" se crean en el espacio de nombres default, mientras que los de "sistema" se encuentran en kube-system.

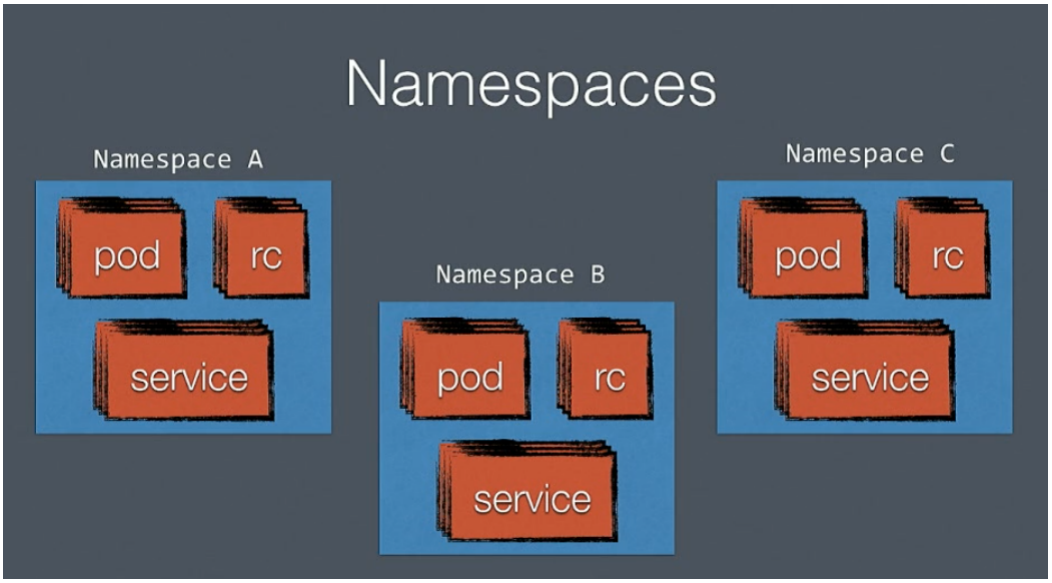


Figura 3.19: Espacios de nombres.

3.10.6. Autoescalado de POD's

Con el autoescalado horizontal de pod, Kubernetes automáticamente escala el numero de pods, dependiendo del uso de CPU (o cualquier otra métrica de las soportadas) que hayamos definido en nuestro deployment.

Escala automáticamente el número de pods en un controlador de replicación, implementación o conjunto de réplicas en función de la utilización observada de la CPU (o, con el soporte de métricas personalizadas, en otras métricas proporcionadas por la aplicación). El autoescalado horizontal de pods no se aplica a los objetos que no se pueden escalar.

El autoescalado horizontal de pod se implementa como un recurso de API de Kubernetes y un controlador. El recurso determina el comportamiento del controlador. El controlador ajusta periódicamente la cantidad de réplicas en un controlador de replicación o implementación para que coincida con el uso promedio observado de la CPU con el destino especificado por el usuario.

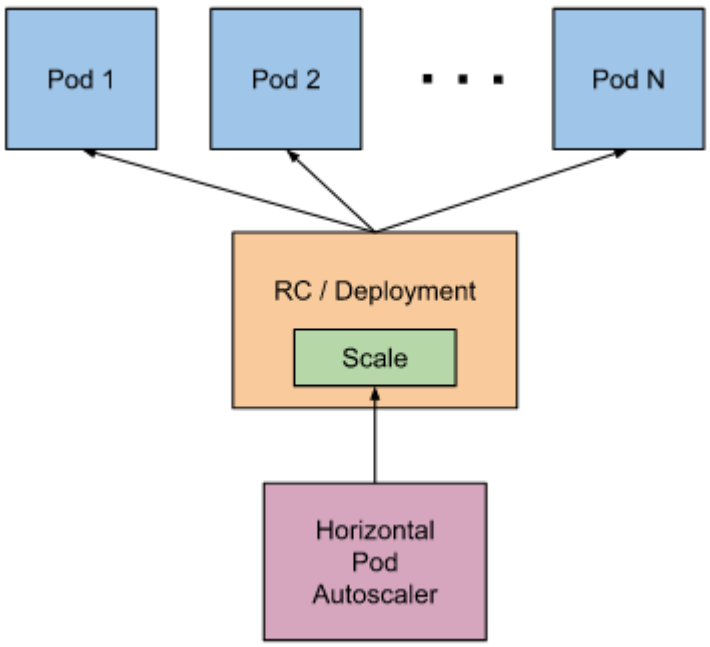


Figura 3.20: Funcionamiento autoescalado horizontal de pods.

3.11. Flannel

Es una forma sencilla y fácil de configurar un tejido de red de tres capas diseñado para Kubernetes.

Flannel se centra en la creación de redes. Ejecuta un pequeño agente binario único llamado flanneld en cada host y es responsable de asignar una concesión de subred a cada host de un espacio de direcciones más grande y preconfigurado. Flannel utiliza la API de Kubernetes o etcd directamente para almacenar la configuración de la red, las subredes asignadas y cualquier información auxiliar. Los paquetes se reenvían utilizando uno de varios mecanismos de backend, incluyendo VXLAN y varias integraciones en la nube.

Las plataformas como Kubernetes asumen que cada contenedor tiene una IP única y enrutable dentro del cluster. La ventaja de este modelo es que elimina las complejidades de mapeo de puertos que surgen al compartir una única IP de host.

Flannel es responsable de proporcionar una red IPv4 de capa 3 entre varios nodos en un cluster. Flannel no controla cómo se conectan en red los contenedores al host, solo cómo se transporta el tráfico entre los hosts. Sin embargo, la flannel proporciona un complemento CNI para Kubernetes y una guía sobre la integración con Docker.

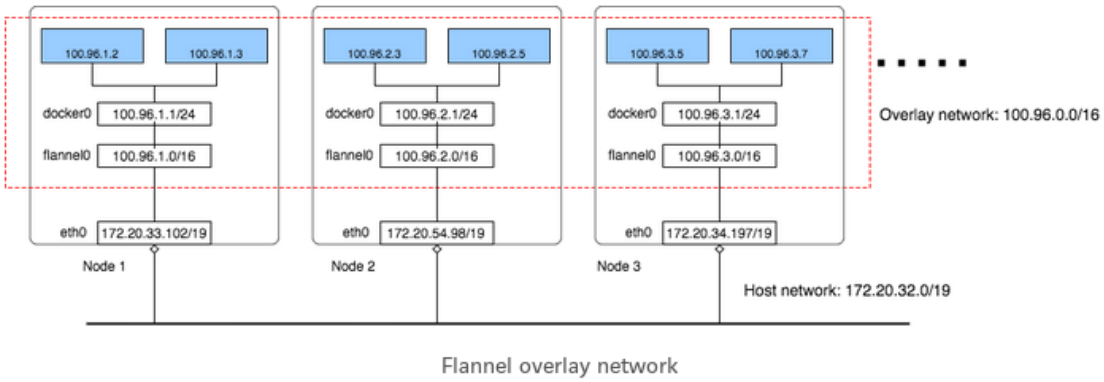


Figura 3.21: Superposición de capas flannel.

Capítulo 4

Desarrollo e implementación

En este capítulo serán expuestos todos los pasos necesarios ha realizar para la creación de los componentes requeridos que conforman el sistema final. Los pasos realizados para la obtención de este sistema han sido:

- Implementación de la **API REST KubeKVM App** que dota la sistema de un servicio entre la base de datos y el script KVM que requerirá información de la misma a través de un frontal Web.
- Implementación del **script KVM** que funcionando como un demonio consume los datos de los despliegues almacenados en la base de datos a través de la API REST KubeKVM App y realiza la creación de las máquinas virtuales necesarias para cada despliegue utilizando la tecnología de virtualización KVM.
- Implementación de los archivos **Dockerfile** de **Docker** y **yaml** de **Kubernetes** para realizar la exposición de los servicios ofrecidos por la **API REST KubeKVM App** a través de **Kubernetes**.

4.1. API REST KubeKVM App

En esta sección se explica como se ha procedido al desarrollo de la API REST KubeKVM, es decir se implementará una aplicación web de notas que contará con backend y frontend, utilizando CRUD de Node.js, una base de datos MongoDB, el framework Express de Node.js y otras tecnologías de JavaScript como CSS, Bootstrap para estilizar nuestra aplicación.

Uno de los primeros pasos es realizar la configuración inicial del sistema para implementar la API, dicha configuración se encuentra en el anexo con título **Anexo I: Configuración inicial para el desarrollo de la API**.

Tras haber realizado todos los pasos anteriores descritos en el anexo, ya disponemos del equipo configurado y listo para el desarrollo de esta API REST. Lo primero que debemos hacer el crear la estructura de carpeta que almacenarán los diferentes archivos y clases que formarán parte de la API.

Dentro de la carpeta del proyecto, crearemos una carpeta llamada **src** que contendrá todo el código, tanto el código de las vistas del frontend como el código backend del servidor. En su interior crearemos:

- Un archivo llamado **index.js** que será el archivo principal de toda nuestra aplicación.
- Un archivo llamado **database.js** que será el que establezca la conexión con la base de datos.
- Un fichero de configuración llamado **config.js** para facilitar la modificación de ciertas variables.
- La carpeta **views** que almacenará todos los archivos que vamos a enviar al navegador, es decir todas las vistas HTML.
- La carpeta **routes** que permitirá crear las URLs o rutas de nuestro servidor.

- La carpeta **public** que permitirá insertar todos los archivos estáticos como imágenes, fuentes, archivos CSS, archivos JavaScript, ...
- La carpeta **models** que permitirá definir como van a lucir los datos que se quieren insertar en la base de datos.

En la imagen siguiente se puede observar cual es la disposición de los componentes del proyecto tras la creación de su estructura inicial.

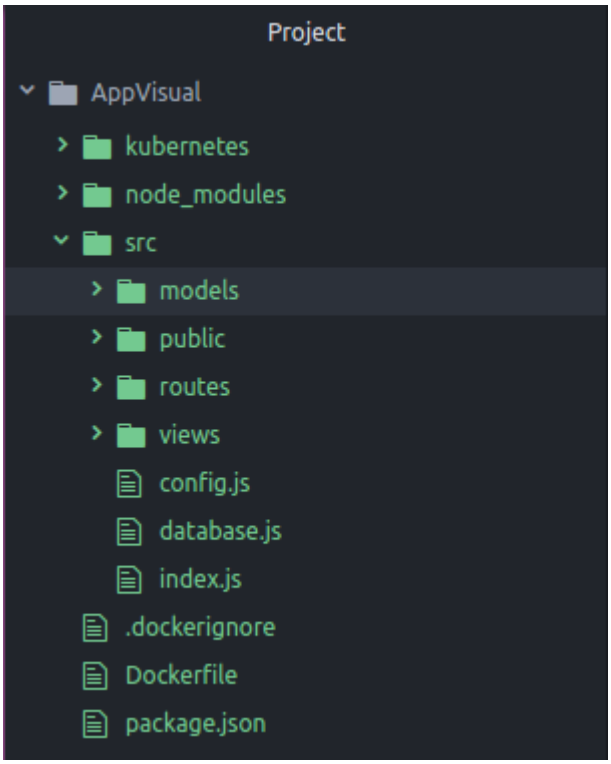


Figura 4.1: Estructura del proyecto.

4.1.1. Implementación inicial del backend del servidor

En este apartado se realiza la configuración inicial del backend. El primer paso es configurar la clase **config.js**, con la variable del número de **puerto** por el que escuchará el servidor que será una variable de entorno recibida llamada **process.env.PORT** o el número de puerto especificado, en este caso el **3000**.

```
module.exports = {
        port: process.env.PORT || 3000,
}
```

El uso de **module.exports** permite que el código pueda ser importado como un módulo en otras clases mediante el uso de **require**.

Antes de comenzar, vamos a estructurar el código de la clase **index.js** en varías secciones para hacerlo más sencillo de manera que tendremos:

```
Importacion de modulos
...
...
//Settings
Aqui iran todas nuestras configuraciones
...
//Middlewares
Aqui iran todas nuestras funciones que van a ser ejecutadas antes de
   llegar al servidor o cuando lleguen al servidor, antes de ser
   pasados a las rutas
...
//Global Variables
Aqui iran las variables globales, es decir ciertos datos que queremos
   que sean accesibles para toda la aplicacion
...
//Routes
Aqui iran las rutas utilizadas por la aplicacion
...
//Static Files
Aqui es donde se configura donde estara la carpeta de archivos
   estaticos
...
//Server is listening
Aqui es donde se inicializa el servidor
```

```
...
```

Tras esto, procedemos a implementar el código del servidor en la clase **index.js** con las funciones principales para crear la conexión, cerrarla , enviar y recibir mensajes. Para ello importamos los módulos necesarios que son **express** y **config**.

```
Importacion de modulos
const express = require('express');
const config = require('./config');
...
```

Ejecutamos la función **express**, lo guardamos en una constante **app**, creamos una configuración del puerto utilizando la variable **port** del módulo **config**, iniciamos el servidor mediante **app.listen** y le pasamos el número de puerto mediante **app.get('port')**, dentro de la función introducimos el **console.log** para ver el mensaje en el terminal.

```
...
//Initializations
const app = express();
//Settings
app.set('port', config.port);
...
//Server is listenning
app.listen(app.get('port'), () => {
        console.log('Server on port', app.get('port'))
});
```

De esta manera ya podemos realizar la conexión inicial con el servidor. Ahora procedemos a implementar las **rutas** para nuestra página principal y dotar a esta de un mecanismo para la recepción de verbos HTTP, en este caso el método **GET**. Comenzamos creando un archivo llamado **/routes/index.js**. Tras esto, vamos al **index.js** principal y llamamos a esta ruta mediante **app.use**.

```
...
//Routes
app.use(require('./routes/index'));
...
```

A continuación, implementamos la clase **/routes/index.js**, utilizando el módulo **express** para crear rutas, mediante el método **Router** incluido en el módulo. Con el uso de este método podremos recibir la petición **GET**, cuando se llame a la página principal de la aplicación, mediante **router.get('/')** se manejará a través de una función que maneje las peticiones y las respuestas **(req, res)** y de momento devolverá un mensaje que diga Index utilizando **res.send** . Para terminar exportaremos esa ruta con el uso de **module.exports** para que pueda ser llamada desde el **index.js**.

```
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
        res.send('Index');
});

module.exports = router;
```

De la misma manera que hemos realizado la implementación de las rutas para la página principal, creamos otro archivo llamado **routes/deployments.js** en el que se realizara la recepción de los verbos HTTP sobre los diferentes despliegues almacenados en la base de datos como se verá más adelante. De momento vamos al **index.js** principal y llamamos a esta ruta mediante **app.use**.

```
...
//Routes
app.use(require('./routes/deployments'));
app.use(require('./routes/index'));
...
```

Hacemos una implementación básica de la clase **/routes/deployments.js** de la misma manera que se realizo con /routes/index.js, exceptuando que esta se visualizara cuando el usuario introduzca la ruta **/deployments**.

```
const express = require('express');
const router = express.Router();

router.get('/deployments', (req, res) => {
        res.send('Deployments');
});
```

```
module.exports = router;
```

Ahora vamos a configurar los ficheros estáticos en la clase principal **index.js**, que estarán en la carpeta public creada anteriormente. Node debe saber donde esta la carpeta public, para hacerlo primero necesitamos importar el módulo **path**, tras esto utilizamos **app.use** para empezar a configurar los archivos estáticos desde express mediante el método **express.static** al que vamos a darle una dirección con el método **path.join** que nos permite unir directorios, a este método le pasamos la constante de node **dirname** que nos devuelve la ruta de donde se esta ejecutando determinado archivo en este caso index.js y esta ruta la concatenamos la carpeta llamada **public**.

```
Importacion de modulos
const express = require('express');
const config = require('./config');
const path = require('path');
...
//Static Files
app.use(express.static(path.join(__dirname, 'public')));
...
```

Para terminar esta configuración inicial del backend del servidor, vamos a realizar la conexión con la base de datos **MongoDB**. Comenzamos modificando la clase **config.js**, con la variable de la dirección de la base de datos que será una variable de entorno recibida llamada **process.env.MONGODB** o la dirección especificada, en este caso **mongodb://localhost:27017/deploymentsKVM**.

```
module.exports = {
        port: process.env.PORT || 3000,
        db: process.env.MONGODB || 'mongodb://localhost:27017/
            deploymentsKVM'
}
```

Seguimos trabajando con la clase **database.js** donde importamos los módulos **mongoose** y **config**. Establecemos la configuración para el funcionamiento de la biblioteca mediante el método **mongose.set** paśandole **useCreateIndex**. Utilizamos el método **mongoose.connect** que nos permite conectarnos a una dirección y le pasamos la dirección **config.db** importada del módulo config, a continuación le colocamos el objeto **useNewUrlParser**. Una vez que se conecte a la base de datos se muestra un mensaje en el terminal mediante **return console.log**, en caso contrario se retorna el error mediante un mensaje en el terminal.

```
const mongoose = require('mongoose');
const config = require('./config');

mongoose.set('useCreateIndex', true);

mongoose.connect(config.db, { useNewUrlParser: true }, (err, res) => {
        if (err) {
                return console.log(`Error al conectar a la base de
                    datos: ${err}`);
        }
                console.log('Conexion a la base de datos establecida...
                    ');
})
```

Para ejecutar esta conexión con la base de datos tenemos que llamarla desde la clase principal **index.js** inicializando la base de datos mediante el uso de **require**.

```
...
//Initializations
const app = express();
require('./database');
...
```

Tras esto ya tenemos el servidor inicializado y la base de datos conectada.

4.1.2. Implementación de las vistas del servidor

En este apartado pasamos a implementar las vistas. Vamos a dejar todos los archivos que son necesarios para enviar al frontend, es decir en lugar de enviar mensajes simplemente de nuestras rutas, vamos a enviarle algunos archivos HTML, en nuestro caso utilizaremos **handlebars**.

Inicialmente trabajamos con la clase principal **index.js**, al igual que hicimos anteriormente con la carpeta public, debemos decirle a Node donde está la carpeta views, para hacerlo utilizamos **app.set**, de damos el nombre de la carpeta **views** para establecer las vistas y le decimos donde esta la carpeta con el uso de **path.join** que nos permite unir directorios, a este método le pasamos la constante de node **dirname** que nos devuelve la ruta de donde se esta ejecutando determinado archivo en este caso index.js y esta ruta la concatenamos la carpeta llamada **views**.

```
...
//Settings
app.set('port', config.port);
app.set('views', path.join(__dirname, 'views'));
...
```

Tras esto debemos configurar **handlebars**. Importamos el módulo **express-handlebars**, se configura utilizando **app.engine**, le damos el nombre **.hbs** que es como serán llamados los archivos de las vistas y la constante **exphbs** donde ha sido importado el módulo. Esa constante será una función a la que le pasaremos un objeto de configuración con ciertas propiedades llamadas **defaultLayout, layoutsDir, partialsDir, extname,** que servirán para saber de que manera se van a utilizar las vistas.

Antes de continuar hay que mencionar que en el proyecto vamos a tener muchas vistas que vamos a enviar al navegador y todos esos archivos van a tener cosas en común como por ejemplo una navegación que se va a repetir en todas las vistas, para no estar escribiendo la navegación en todos los archivos, se utiliza una plantilla donde se coloca el diseño principal y luego las partes que van a cambiar, esto se verá con mas claridad mas adelante. Para crearlo vamos a la carpeta **views** y en su interior creamos la carpeta **layouts**. Dentro de esta carpeta creamos un archivo llamado **main.hbs** en el que colocaremos la plantilla principal de toda la aplicación.

Al igual que con los layouts, debemos crear dentro de la carpeta **views** una carpeta denominada **partials** que serán pequeñas partes de HTML que podemos reutilizar en cualquier vista, esto será de utilidad cuando se trabaje con formularios.

Seguimos configurando el motor de express-handlebars:

- **defaultLayout:** le pasamos **main**, el nombre del archivo principal **main.hbs** creado anteriormente.
- **layoutsDir:** debemos decirle al motor express-handlebars donde está la carpeta **layouts**, para hacerlo utilizamos **path.join** que nos permite unir directorios, le pasamos el método **app.get(views)** para que obtenga la dirección de la carpeta views y esta ruta la concatenamos la carpeta llamada **layouts**.
- **partialsDir:** debemos decirle al motor express-handlebars donde está la carpeta **partials** que crearemos ahora en el interior de la carpeta **views** y utilizaremos mas adelante, para hacerlo utilizamos **path.join** que nos permite unir directorios, le pasamos el método **app.get(views)** para que obtenga la dirección de la carpeta views y esta ruta la concatenamos la carpeta llamada **partials**.
- **extname:** debemos decirle que extensión van a tener nuestros archivos que será **.hbs**

De esta manera ya lo tenemos configurado, pero no lo estamos utilizarlo. Para utilizarlo debemos usar la función **app.set** a la que le pasamos **view engine** para configurar el motor de las vistas y el nombre del motor que es **.hbs**.

```
Importacion de modulos
const express = require('express');
const config = require('./config');
const path = require('path');
const exphbs = require('express-handlebars');
...
//Settings
app.set('port', config.port);
app.set('views', path.join(__dirname, 'views'));
app.engine('.hbs', exphbs({
        defaultLayout: 'main',
        layoutsDir: path.join(app.get('views'), 'layouts'),
        partialsDir: path.join(app.get('views'), 'partials'),
        extname: '.hbs'
}));
app.set('view engine', '.hbs');
...
```

Tras esto ya tenemos las vistas preparadas en la clase principal **index.js**. Ahora comenzamos a implementar las vistas, creamos el archivo **/views/index.hbs** que contendrá las vistas de la página principal de la aplicación, también creamos el archivo **/views/test.hbs** que contendrá las vistas de la página donde se chequea el balanceo de carga de la aplicación. Para que estos archivos sean enviados a la ruta determinada, vamos a **/routes/index.js** y en lugar de enviarle un texto al método **router.get**, le enviamos el archivo **index.hbs** utilizando **res.render** pasándole el archivo **index**, así le estamos diciendo que como respuesta renderice el archivo **index.hbs**. Importamos el módulo **os** que proporciona métodos relacionados con el sistema operativo, de manera similar al anterior creamos otro método **router.get**, cuando el usuario introduzca en el navegador la dirección **/test**, obtenemos el nombre del equipo mediante el método **os.hostname**, le enviamos el archivo **test.hbs** utilizando **res.render** pasándole el archivo **test** y el nombre del equipo, así le estamos diciendo que como respuesta renderice el archivo **test.hbs**.

```
...
var os = require("os");

router.get('/', (req, res) => {
        res.render('index');
```

```
});

router.get('/test', (req, res) => {
        const name = os.hostname();
        res.render('test', {name});
});
...
```

Realizamos la configuración de la vista **/views/index.hbs** a la que mas adelante le daremos un estilo utilizando Bootstrap. De momento solo tendrá un texto.

```
<h1>Index</h1>
```

De manera similar configuramos la vista **/views/test.hbs**.

```
<h1>Test</h1>
```

Para que esta configuración sea efectiva, antes debemos configurar la plantilla **/views/layouts/main.hbs** que creamos anteriormente como plantilla principal para todas la páginas. En ella escribiremos una estructura básica de **HTML**, en la sección de la cabecera que cambiaremos el título por el de la aplicación **KubeKVM App**, en la sección del cuerpo hay dos partes que representen a cosas comunes de todas las vistas que son la navegación llamada **Navigatión** y el píe de página llamado **Footer**. Todas las páginas van a tener la misma vista y lo único que va a cambiar en ellas es lo que esté debajo de Navigation que estará acotado por la sintaxis de handlebars **{{{ body }}}**, así le estamos diciendo que lo único que va a cambiar es lo que esta dentro de las tres llaves. Esta es una de las razones por las que estamos utilizando handlebars.

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8">
                <title>KubeKVM App</title>
        </head>
        <body>

                <p>Navigation</p>

                {{{ body }}}
```

```
        </body>
</html>
```

Ahora es el momento de darle estilo a la vistas **/views/index.hbs** y **/views/test.hbs**. Para esto utilizaremos el framework de **CSS** de Bootstrap. Primero debemos introducir información del CSS de Boostrap en la cabecera de la plantilla **/views/layouts/main.hbs** utilizando una etiqueta **link**.

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8">
                <title>KubeKVM App</title>
                <!-- BOOTSTRAP 4 -->
                <link rel="stylesheet" href="https://stackpath.
                    bootstrapcdn.com/bootstrap/4.2.1/css/bootstrap.min.
                    css" integrity="sha384-
                    GJzZqFGwb1QTTN6wy59ffF1BuGJpLSa9DkKMp0DgiMDm4iYMj70gZWKYbI706tWS
                    " crossorigin="anonymous">
        </head>
        <body>

                <p>Navigation</p>

                {{{ body }}}

                <p>Footer</p>

        </body>
</html>
```

Tras esto podemos darle estilo a la vistas **/views/index.hbs** y **/views/test.hbs**. En la vista **index.hbs** se utiliza una clase **jumbotron** que cuenta con:

- Un título **display-4 text-center** con el nombre de la aplicación junto a una imagen del logo de la Universidad de Extremadura con la etiqueta **logo1** descargada previamente en la carpeta public/img.
- Un texto de introducción de ejemplo **lead**.

- Una linea horizontal **my-4**.
- Un párrafo de ejemplo.
- Un botón **btn btn-primary btn-lg** con una etiqueta que redirecciona a un sitio web.

```
<div class="jumbotron mt-4">
        <h1 class="display-4 text-center">KubeKVM App
                <img src="/img/UNEX.png" class="logo1" alt="LogoUnex">
        </h1>
        <p class="lead">Lorem ipsum dolor sit amet, consectetur
            adipisicing elit.</p>
        <hr class="my-4">
        <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
            sed do eiusmod tempor incididunt ut labore et dolore magna
            aliqua. Ut enim ad minim veniam, quis nostrud exercitation
            ullamco laboris nisi ut aliquip ex ea commodo consequat.
            Duis aute irure dolor in reprehenderit in voluptate velit
            esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
            occaecat cupidatat non proident, sunt in culpa qui officia
            deserunt mollit anim id est laborum.</p>
        <a href="http://google.es" class="btn btn-primary btn-lg">
            Google</a>
</div>
```

En la vista **test.hbs** se utiliza:

- Una clase tarjeta **card**.
- El cuerpo de la tarjeta **card-body** que tiene en el interior:
  - Un texto en forma de título.
  - Una línea horizontal **my-4**.
  - Un párrafo donde se muestra el dato recibido que es el nombre del equipo en el que se esta ejecutando la aplicación.

```
<div class="card">
        <div class="card-body text-center">
                <h1>Test para comprobar el balanceo de carga en el
                    cluster de Kubernetes</h1>
                <hr class="my-4">
                <p>Hola mundo desde el Pod: {{name}}</p>
        </div>
</div>
```

Antes de continuar, si queremos que todo el contenido de la página que se visite esté centrado debemos configurar el cuerpo de la plantilla **/views/layouts/-main.hbs**. Introducimos una etiqueta **main** que es la que define el contenido principal de la aplicación, con la clase de Bootstrap **container** y dentro introducimos el **{{{ body }}}**.

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8">
                <title>KubeKVM App</title>
                <!-- BOOTSTRAP 4 -->
                <link rel="stylesheet" href="https://stackpath.
                    bootstrapcdn.com/bootstrap/4.2.1/css/bootstrap.min.
                    css" integrity="sha384-
                    GJzZqFGwb1QTTN6wy59ffF1BuGJpLSa9DkKMp0DgiMDm4iYMj70gZWKYbI706tWS
                    " crossorigin="anonymous">
        </head>
        <body>

                <p>Navigation</p>

                <main class="container p-2">
                        {{{ body }}}
                </main>

        </body>
</html>
```

Para finalizar el trabajo con las vistas vamos a configurar los **estilos CSS** para no tener que utilizar los que nos da Bootstrap. Dentro de la carpeta **public**, creamos la carpeta **css** y en el interior un archivo llamado **main.css**. Dentro del archivos vamos a darle un color utilizando la etiqueta **body** y también estableceremos el ancho del **logo1** perteneciente la vista situada en el archivo **/views/index.hbs**.

```
body{
        background: #1488CC;  /* fallback for old browsers */
        background: -webkit-linear-gradient(to right, #2B32B2, #1488CC)
            ;  /* Chrome 10-25, Safari 5.1-6 */
        background: linear-gradient(to right, #2B32B2, #1488CC); /* W3C
            , IE 10+/ Edge, Firefox 16+, Chrome 26+, Opera 12+, Safari
            7+ */
}

.logo1{
        width: 10%;
}
```

Para que se aplique este color en todas las plantillas tenemos que llamarlo en nuestra plantilla principal **/views/layouts/main.hbs** utilizando una etiqueta **link**.

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8">
                <title>KubeKVM App</title>
                <!-- BOOTSTRAP 4 -->
                <link rel="stylesheet" href="https://stackpath.
                    bootstrapcdn.com/bootstrap/4.2.1/css/bootstrap.min.
                    css" integrity="sha384-
                    GJzZqFGwb1QTTN6wy59ffF1BuGJpLSa9DkKMp0DgiMDm4iYMj70gZWKYbI706tWS
                    " crossorigin="anonymous">
                <!-- CUSTOM CSS -->
                <link rel="stylesheet" href="/css/main.css">
        </head>
        <body>

                <p>Navigation</p>
```

```
                <main class="container p-2">
                        {{{ body }}}
                </main>

        </body>
</html>
```

De esta manera ya tenemos implementadas inicialmente las vistas en el servidor, aunque más adelante cuando se haya realizado la implementación de las funciones CRUD en el servidor se implementarán las vistas referentes a los despliegues almacenados en la base de datos.

4.1.3. Implementación de funciones CRUD en el servidor

En esta apartado se crearán las rutas relacionadas con los despliegues, las que nos van a permitir obtener un formulario, poder listar, agregar, modificar o editar los despliegues, es decir vamos a crear todas las operaciones relacionadas con los despliegues.

Para comenzar se realizarán las modificaciones en la ruta encargada de esas acciones situada en el archivo **/routes/deployments.js**. Se crea una ruta que permita al usuario ver un formulario, utilizando el método **router.get**, cuando el usuario introduzca en el navegador la dirección **/deployments/add**, le enviaremos el formulario **/views/deployments/newDeployment.hbs** que crearemos a continuación, utilizando **res.render** pasándole el archivo **deployments/newDeployment**, así le estamos diciendo que como respuesta renderice el archivo **newDeployment.hbs**.

```
const express = require('express');
const router = express.Router();

router.get('/deployments/add', (req, res) => {
        res.render('deployments/newDeployment');
});

router.get('/deployments', (req, res) => {
        res.send('Deployments');
});
```

```
module.exports = router;
```

Para crear el formulario, en el interior creamos una carpeta que estará relacionada solo con los despliegues llamada **deployments** y dentro crearemos el formulario para agregar un nuevo despliegue que sera el archivo llamado **newDeployment.hbs**. En el interior crearemos un formulario dentro de una clase columna **col-md-4 mx-auto** que servirá para centrar la tarjeta, esta clase estará compuesta por:

- Una clase tarjeta **card** que tiene en el interior:
  - La cabecera de la tarjeta **card-header text-center** que tiene en el interior:
    - Un texto de título junto a la imagen del logo de Kubernetes con la etiqueta **logo2** descargada previamente en la carpeta public/img.
  - El cuerpo de la tarjeta **card-body** donde irán todos los siguientes componentes del formulario **form** especificando la dirección **/deployments/-newDeployment** a la que se mandarán los datos cuando un usuario los introduzca y la acción a realizar que será un método **POST**, para recibir estos datos a continuación se creará una ruta para recibir los datos mediante el método **router.post** en el archivo **/routes/deployments.js**. Los campos en donde se introducirán estos datos relativos al nuevo despliegue son:
    - Una entrada de texto tipo **input** con el nombre del despliegue.
    - Un desplegable tipo **select** referente al tipo de despliegue.
    - Una entrada de números tipo **input** con el número de réplicas.
    - Un botón tipo **submit** para realizar el envío de los datos.

```
<div class="col-md-4 mx-auto">
        <div class="card">
                <div class="card-header text-center">
                        <h3>Nuevo Despliegue</h3>
                                <img src="/img/kubernetes.png" class="
                                    rounded-circle mx-auto d-block m-3 
                                    logo2" alt="LogoKubernetes">
```

```
                </div>
                <div class="card-body">
                        <form action="/deployments/newDeployment"
                            method="post">
                                <div class="form-group">
                                        <label>Despliegue:</label>
                                        <input type="text" name="name"
                                            class="form-control"
                                            placeholder="Nombre del
                                            despliegue" autofocus>
                                </div>
                                <div class="form-group">
                                        <label>Tipo:</label>
                                        <select name="type" class="form
                                            -control">
                                                <option value="
                                                    RedesNeuronales">
                                                    RedesNeuronales</
                                                    option>
                                                <option value="
                                                    TecnologiaDeComputadores
                                                    ">
                                                    TecnologiaDeComputadores
                                                    </option>
                                        </select>
                                </div>
                                <div class="form-group">
                                        <label>Replicas:</label>
                                        <input type="number" class="
                                            form-control" name="replicas
                                            " value="1" min="1">
                                </div>
                                <div class="form-group">
                                        <button class="btn btn-primary
                                            btn-block" type="submit">
                                                Guardar
                                        </button>
                                </div>
                        </form>
```

```
                </div>
        </div>
</div>
```

Antes de continuar, modificamos los **estilos CSS** en el archivo **/public/css/-main.css** añadiendo el ancho del **logo2** perteneciente esta vista.

```
...
.logo2{
        width: 50%;
}
```

Ahora creamos la ruta **/deployments/newDeployment** en el archivo **/routes/deployments.js** para recibir la petición **POST** mediante el método **router.post**, se manejará a través de una función que maneje las peticiones y las respuestas **(req, res)**, a continuación se introducirán estos datos en la base de datos pero de momento mostrará por el terminal utilizando **console.log** el cuerpo de la petición **req.body** y devolverá un mensaje que diga 'ok' utilizando **res.send**.

```
...
router.post('/deployments/newDeployment', (req, res) => {
        console.log(req.body);
        res.send('ok');
}
...
```

Comenzamos a trabajar con la base de datos de **MongoDB** para poder almacenar los datos introducidos en el formulario. Necesitamos un crear un modelo de datos, para ello vamos a la carpeta **models**, en su interior vamos a definir cuales van a ser los tipos de los datos. Creamos un archivo llamado **deployments.js**, importamos **mongoose** para crear un esquema de datos mediante la funnción **mongoose.Schema** y a definimos el esquema de datos:

- **name** que sera una cadena única y este campo sera requerido para introducir el despliegue en la base de datos.
- **type** que sera una cadena que solo puede tener 2 valores.
- **replicas** que será numérico y por defecto tendrá el valor 1.

- **estado** que será una cadena que solo puede tener 3 valores y por defecto tendrá el valor Pendiente.

Para finalizar con esta implementación exportamos este modelo de datos usando **module.exports** pasándole el nombre **Deployment** y el esquema **DeploymentSchema**. De esta manera ya podemos almacenar un nuevo despliegue, actualizarlo o eliminarlo de la base de datos.

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const DeploymentSchema = Schema({
        name: { type:String, unique:true, required:true},
        type: { type: String, enum: ['TecnologiaDeComputadores', '
            RedesNeuronales'] },
        replicas: { type: Number, default: 1 },
        estado: { type: String, enum: ['Inactivo', 'Pendiente', 'Activo
            '], default: 'Pendiente'}
})

module.exports = mongoose.model('Deployment', DeploymentSchema);
```

Tras esto, importamos el modelo de datos creado en el archivo **/models/deployment.js** y modificamos la ruta **/deployments/newDeployment** creada anteriormente en el archivo **/routes/deployments.js** para recibir la petición **POST** mediante el método **router.post** para que almacene los datos recibidos. Primero extraemos de los datos de la petición el nombre **req.name**, el tipo **req.type** y el número de replicas **replicas**. Para validar los errores creamos un array donde almacenar los mensajes de errores y comprobamos:

- Si no se ha introducido un nombre se introduce en el array un texto de error.

- Si hay errores se vuelve a mostrar el formulario para insertar un despliegue pasándole los errores para que los muestre mediante un mensaje de alerta que implementaremos a continuación, además le pasaremos el nombre, el tipo y el número de replicas para que el usuario no tenga que volver a rellenar esos campos en el formulario.

- Si no hay errores insertamos los datos en la base de datos mediante el método **save** y redireccionamos hacia la ruta **/deployments** manejada por la función **router.get** situada dentro del mismo archivo, que mas adelante se modificará para que consulte todos los despliegues insertados en la base de datos.

```
...
const Deployment = require('../models/deployment');
...
router.post('/deployments/newDeployment', (req, res) => {
        const name = req.body.name;
        const type = req.body.type;
        const replicas = req.body.replicas;
        const errors= [];
        if(!name){
                errors.push({text: 'Por favor escribe un nombre para el
                    despliegue'});
        }
        if(errors.length > 0) {
                res.render('deployments/newDeployment', {
                errors,
                name,
                type,
                replicas
        });
        }else{
                const newDeployment = new Deployment({name, type,
                    replicas});
                newDeployment.save();
                res.redirect('/deployments');
        }
});
...
```

Para mostrar los errores cuando los haya, se mostrarán en la parte superior del formulario alertas de Bootstrap. Para implementarlas las introducimos en la vista correspondiente al formulario en el archivo **/views/deployments/newDeployment.hbs**, pero para poder cerrar esas ventanas de alerta que se crearán antes debemos introducir el framework de **JavaScript** de Bootstrap en la plantilla principal situada en el archivo **/views/layouts/main.hbs** en el interior de la etiqueta **body**.

```
...
<body>

        <p>Navigation</p>

        <main class="container p-2">
                {{{ body }}}
        </main>

        <!-- SCRIPTS -->
        <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
            integrity="sha384-q8i/X+965
            DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
            crossorigin="anonymous"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js
            /1.14.3/umd/popper.min.js" integrity="sha384-ZMP7rVo3mIykV
            +2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIPm49"
            crossorigin="anonymous"></script>
        <script src="https://stackpath.bootstrapcdn.com/bootstrap
            /4.1.3/js/bootstrap.min.js" integrity="sha384-
            ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/
            JmZQ5stwEULTy" crossorigin="anonymous"></script>

</body>
...
```

Ahora implementamos las alertas en la parte superior del archivo **/views/deployments/newDeployment.hbs** de manera que por cada error recibido **each errors** muestra una alerta.

```
{{#each errors}}
        <div class="alert alert-danger alert-dismissible fade show"
            role="alert">
                {{text}}
                <button type="button" class="close" data-dismiss="alert
                    " aria-label="Close">
                        <span aria-hidden="true">&times;</span>
                </button>
        </div>
{{/each}}
...
```

Modificamos la ruta **/deployments** en el interior del archivo **/routes/deployments.js** para atender a una petición **GET** utilizando el método **router.get** que mostrará todos los despliegues almacenados en la base de datos utilizando la consulta **find**, dependiendo de la respuesta de la consulta:

- Si hay un error en la consulta, devuelve el código HTTP **500 Internal Server Error** referente a situaciones de error ajenas a la naturaleza del servidor web.
- Si no hay ningún despliegue almacenado en la base de datos, devuelve el código HTTP **404 Not Found** referente a que el servidor web no encuentra el recurso solicitado.
- Si se realiza la consulta correctamente, se utiliza **res.render** pasándole la vista correspondiente a todos los despliegues almacenados en la base de datos almacenada en el archivo **/views/deployments/allDeployments.hbs** que se implementara a continuación. Al igual que se realizó anteriormente con los errores, a esta vista se le pasarán los despliegues para que pueda mostrarlos.

```
...
router.get('/deployments', (req, res) => {
        Deployment.find({}, (err, deployments) =>{
                if (err) return res.status(500).send({message: `Error
                    al realizar la peticion: ${err}`});
```

```
                if(!deployments) return res.status(404).send({message:
                    'No existen despliegues'});

                res.render('deployments/allDeployments', {deployments})
                    ;
        });
});
...
```

Implementamos la vista situada en el archivo **/views/deployments/allDeployments.hbs** que corresponde a todos los despliegues almacenados en la base de datos de manera que por cada despliegue recibido se mostrará una objeto de clase tarjeta con sus datos con el nombre, el tipo, el estado del despliegue y el numero del replicas del mismo. Las tarjetas se implementarán dentro de una clase fila **row** esta clase estará compuesta por:

- Un **each deployments** que recorre todos los despliegues recibidos, por cada despliegue:
    - Columnas **col-md-3** donde se ubicarán las diferentes tarjetas compuestas por:
        - Una clase tarjeta **card** que contiene:
            - Un cuerpo de la tarjeta **card-body** compuesto por: un título **tittle** con el nombre del despliegue recibido junto a un botón **fas fa-edit** cuya ruta **/deployments/edit/{{_id}}** se implementará mas adelante para modificar los campos del despliegue con la id del despliegue recibido mediante un formulario, una línea horizontal **my-4** para separar los campos, el tipo del despliegue recibido, el estado del despliegue recibido, el número de replicas del despliegue recibido y un botón **btn btn-danger btn-block btn-sm** para eliminar el despliegue cuya ruta se implementará mas adelante.

- Un **else** para en el caso en el que no haya despliegues almacenados en la base de datos se mostrará un botón **btn btn-success btn-block** que redirecciona a la ruta **/deployments/add** donde se crea un nuevo despliegue a través del formulario.

```
<div class="row">
        {{#each deployments}}
                <div class="col-md-3">
                        <div class="card">
                                <div class="card-body">
                                        <h4 class="card-title d-flex
                                            justify-content-between
                                            align-items-center">
                                                {{name}} <a href="/
                                                    deployments/edit/{{
                                                    _id}}"><i class="fas
                                                    fa-edit"></i></a>
                                        </h4>
                                        <hr class="my-4">
                                        <p>{{type}}</p>
                                        <p>{{estado}}</p>
                                        <p>{{replicas}}</p>
                                        <button class="btn btn-danger
                                            btn-block btn-sm" type="
                                            submit">
                                                Eliminar
                                        </button>
                                </form>
                        </div>
                </div>
        </div>
        {{else}}
                <div class="card mx-auto">
                        <div class="card-body">
                                <p class="lead">Actualmente no hay
                                    ningun despliegue creado.</p>
                                <a href="/deployments/add" class="btn
                                    btn-success btn-block">Crea un nuevo
                                     despliegue</a>
```

```
                        </div>
                    </div>
            {{/each}}
</div>
```

Añadimos a la plantilla principal **/views/layouts/main.hbs** la biblioteca para iconos dentro de la etiqueta **head**.

```
<head>
        ...
        <!-- FONT AWESOME -->
        <link rel="stylesheet" href="https://use.fontawesome.com/
            releases/v5.5.0/css/all.css" integrity="sha384-
            B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+
            L3BlXeVIU" crossorigin="anonymous">
        ...
</head>
```

Implementamos la ruta del botón creado anteriormente para modificar los despliegues de la base de datos. Creamos la ruta **/deployments/edit/:id** en el interior del archivo **/routes/deployments.js** para atender a una petición **GET** utilizando el método **router.get** que modificará un despliegue almacenado en la base de datos con el **id** correspondiente utilizando la consulta **findById**, donde el valor del campo **id** es un valor único otorgado por la base de datos al realizar la inserción de un nuevo despliegue. Dependiendo de la respuesta de la consulta:

- Si hay un error en la consulta, devuelve el código HTTP **500 Internal Server Error** referente a situaciones de error ajenas a la naturaleza del servidor web.
- Si no hay ningún despliegue almacenado en la base de datos, devuelve el código HTTP **404 Not Found** referente a que el servidor web no encuentra el recurso solicitado.
- Si se realiza la consulta correctamente, se utiliza **res.render** pasándole la vista correspondiente al formulario almacenado en el archivo **/views/deployments/editDeployment.hbs** que se implementara a continuación. A esta vista se le pasarán los despliegues para que muestre los valores de sus campos en el formulario.

```
...
router.get('/deployments/edit/:id', (req, res) => {
        Deployment.findById(req.params.id, (err, deployment) => {
                if (err) return res.status(500).send({message: `Error
                    al realizar la peticion: ${err}`});
                if(!deployment) return res.status(404).send({message: '
                    El despliegue no existe'});

                res.render('deployments/editDeployment', {deployment});
        });
});
...
```

Antes de implementar la vista del formulario, necesitamos que los formularios puedan utilizar otros métodos HTTP PUT y DELETE además de GET y POST, para esto debemos importar el módulo **method-override** en la clase principal de la aplicación **index.js** y utilizarla mediante el método **app.use** al que se le pasará el input oculto **_method**.

```
...
const methodOverride = require('method-override');
...
//Middlewares
app.use(methodOverride('_method'));
...
```

Para la vista del formulario, creamos el archivo **/views/deployments/editDeployment.hbs**. En el interior crearemos un formulario dentro de una clase columna **col-md-4 mx-auto** que servirá para centrar la tarjeta, esta clase estará compuesta por:

- Una clase tarjeta **card** que tiene en el interior:
    - La cabecera de la tarjeta **card-header** con un título.

- El cuerpo de la tarjeta **card-body** donde irán todos los siguientes componentes del formulario **form** especificando la dirección **/deployments/editDeployment/{{deployment._id}}?_method=put** a la que se mandarán los datos cuando un usuario los introduzca y la acción a realizar que será un método **PUT** creando debajo un input oculto **hidden** con el nombre **_method** que se colocará como una consulta oculta **?_method=put** al final de la dirección, para recibir estos datos a continuación se creará una ruta para recibir los datos mediante el método **router.put** en el archivo **/routes/deployments.js**. Los campos del formulario que contienen los datos del despliegue recibidos son:
  - Una entrada de texto tipo **input** con el nombre del despliegue.
  - Un desplegable tipo **select** referente al tipo de despliegue.
  - Un desplegable tipo **select** referente al estado de despliegue.
  - Una entrada de números tipo **input** con el número de réplicas.
  - Un botón tipo **submit** para realizar el envío de los datos.

```
<div class="col-md-4 mx-auto">
        <div class="card">
                <div class="card-header">
                        <h3>Editar Despliegue</h3>
                </div>
                <div class="card-body">
                        <form action="/deployments/editDeployment/{{
                            deployment._id}}?_method=put" method="post">
                                <input type="hidden" name="_method"
                                    value="put">
                                <div class="form-group">
                                        <input type="text" name="name"
                                            class="form-control"
                                            placeholder="Nombre del
                                            despliegue" value="{{
                                            deployment.name}}">
                                </div>
                                <div class="form-group">
                                        <label>Tipo:</label>
```

```
                                    <select name="type" class="form
                                        -control">
                                            <option selected="true"
                                                disabled="disabled"
                                                value="{{deployment
                                                .type}}">{{
                                                deployment.type}}
                                            <option value="
                                                RedesNeuronales">
                                                RedesNeuronales</
                                                option>
                                            <option value="
                                                TecnologiaDeComputadores
                                                ">
                                                TecnologiaDeComputadores
                                                </option>
                                    </select>
                                </div>
                                <div class="form-group">
                                    <label>Estado:</label>
                                    <select name="estado" class="
                                        form-control">
                                            <option selected="true"
                                                disabled="disabled"
                                                value="{{deployment
                                                .estado}}">{{
                                                deployment.estado}}
                                            <option value="Inactivo
                                                ">Inactivo</option>
                                            <option value="
                                                Pendiente">Pendiente
                                                </option>
                                    </select>
                                </div>
                                <div class="form-group">
                                    <label>Replicas:</label>
                                    <input type="number" class="
                                        form-control" name="replicas
                                        " value="{{deployment.
```

```
                                        replicas}}" min="1">
                                </div>
                                <div class="form-group">
                                        <button class="btn btn-primary
                                            btn-block" type"submit">
                                                Guardar
                                        </button>
                                </div>
                        </form>
                </div>
        </div>
</div>
```

Creamos la ruta **/deployments/editDeployment/:id** en el interior del archivo **/routes/deployments.js** para atender a una petición **PUT** utilizando el método **router.post** que almacenará el despliegue modificado en la base de datos con el **id** correspondiente utilizando la consulta **findByIdAndUpdate**. Dependiendo de la respuesta de la consulta:

- Si hay un error en la consulta, devuelve el código HTTP **500 Internal Server Error** referente a situaciones de error ajenas a la naturaleza del servidor web.
- Si no hay ningún despliegue con esa id almacenado en la base de datos, devuelve el código HTTP **404 Not Found** referente a que el servidor web no encuentra el recurso solicitado.
- Si se realiza la consulta correctamente y se actualizan los datos sin errores, se redirecciona hacia la ruta **/deployments** manejada por el método **router.get** situada dentro del mismo archivo, que muestra todos los despliegues almacenados en la base de datos.

```
...
router.put('/deployments/editDeployment/:id', (req, res) => {
        Deployment.findByIdAndUpdate(req.params.id, req.body, (err,
            deploymentUpdated) => {
                if (err) return res.status(500).send({message: `Error
                    al actualizar el despliegue: ${err}`});
                if(!deploymentUpdated) return res.status(404).send({
                    message: 'El despliegue no existe'});
```

```
                res.redirect('/deployments');
        });
});
...
```

Antes de implementar el botón creado para eliminar los despliegues de la base de datos, debemos modificar la vista situada en el archivo **/views/deployments/allDeployments.hbs** para introducir un input oculto **hidden** de la misma manera que se ha realizado anteriormente. En el cuerpo de la tarjeta **card-body** a continuación de las réplicas, se implementa este botón dentro del formulario **form** especificando la dirección **/deployments/delete/{{_id}}?_method=delete**, la acción a realizar que será un método **DELETE** creando debajo un input oculto **hidden** con el nombre **_method** que se colocará como una consulta oculta **?_method=delete** al final de la dirección, para recibir estos datos a continuación se creará una ruta para recibir los datos mediante el método **router.delete** en el archivo **/routes/deployments.js**.

```
...
<p>{{replicas}}</p>
<form action="/deployments/delete/{{_id}}?_method=delete" method="post"
    >
        <input type="hidden" name="_method" value="delete">
        <button class="btn btn-danger btn-block btn-sm" type="submit">
                Eliminar
        </button>
</form>
...
```

Ahora implementamos la ruta del botón creado anteriormente para eliminar los despliegues de la base de datos. Creamos la ruta **/deployments/delete/:id** en el interior del archivo **/routes/deployments.js** para atender a una petición **DELETE** utilizando el método **router.delete** que elimina un despliegue almacenado en la base de datos con el **id** correspondiente utilizando la consulta **findById**, donde el valor del campo **id** es un valor único otorgado por la base de datos al realizar la inserción de un nuevo despliegue. Dependiendo de la respuesta de la consulta:

- Si hay un error en la consulta, devuelve el código HTTP **500 Internal Server Error** referente a situaciones de error ajenas a la naturaleza del servidor web.

- Si no hay ningún despliegue almacenado en la base de datos, devuelve el código HTTP **404 Not Found** referente a que el servidor web no encuentra el recurso solicitado.
- Si se realiza la consulta correctamente, se utiliza la función **deployment.remove** con el despliegue recibido al realizar la consulta en la base de datos. Dependiendo de la respuesta recibida por esta función:
  - Si la función devuelve un error, devuelve el código HTTP **500 Internal Server Error** referente a situaciones de error ajenas a la naturaleza del servidor web.
  - Si la función realiza la operación correctamente y se eliminan los datos sin errores, se redirecciona hacia la ruta **/deployments** manejada por el método **router.get** situada dentro del mismo archivo, que muestra todos los despliegues almacenados en la base de datos.

```
...
router.delete('/deployments/delete/:id', (req, res) => {
        Deployment.findById(req.params.id, (err, deployment) => {
                if (err) return res.status(500).send({message: `Error
                    al borrar el despliegue: ${err}`});
                if(!deployment) return res.status(404).send({message: '
                    El despliegue no existe'});
                deployment.remove(err => {
                        if (err) return res.status(500).send({message:
                            `Error al borrar el despliegue: ${err}`});
                        req.flash('success_msg', 'Despliegue Eliminado
                            Satisfactoriamente');
                        res.redirect('/deployments');
                })
        })
})
```

De manera complementaria a la ruta en la que se obtienen todos los datos de los despliegues almacenados en la base de datos, creamos la ruta **/deploymentsScript** en el interior del archivo **/routes/deployments.js** para atender a una petición **GET** realizada por el script KVM que se implementará en la siguiente sección, utilizando el método **router.get** que enviará como respuesta utilizando la función **res.send** todos los despliegues almacenados en la base de datos utilizando la consulta **find**, dependiendo de la respuesta de la consulta:

- Si hay un error en la consulta, devuelve el código HTTP **500 Internal Server Error** referente a situaciones de error ajenas a la naturaleza del servidor web.
- Si no hay ningún despliegue almacenado en la base de datos, devuelve el código HTTP **404 Not Found** referente a que el servidor web no encuentra el recurso solicitado.
- Si se realiza la consulta correctamente, se utiliza **res.send** devolviendo el código HTTP 200 relativo a una consulta correcta y los datos de todos los despliegues almacenados en la base de datos.

```
...
router.get('/deploymentsScript', (req, res) => {
        Deployment.find({}, (err, deployments) =>{
                if (err) return res.status(500).send({message: `Error
                    al realizar la peticion: ${err}`});
                if(!deployments) return res.status(404).send({message:
                    'No existen despliegues'});

                res.status(200).send({deployments: deployments});
        });
});
...
```

Para terminar esta sección. vamos a implementar unos mensajes para mostrar que los datos relativos a los despliegues están siendo almacenados, modificados o eliminados. Importamos el módulo **connect-flash** en la clase principal de la aplicación **index.js** que nos permitirá enviar mensajes entre múltiples vistas de la aplicación. Se utiliza mediante el método **app.use** y a continuación para que todas las vistas tengan acceso a esos mensajes creamos dos variables globales llamadas **res.locals.success_msg** y **res.locals.error_msg** que almacenan esos mensajes **flash**.

```
...
const flash = require('connect-flash');
...
//Middlewares
app.use(methodOverride('_method'));
app.use(flash());

//Global Variables
app.use((req, res, next) => {
        res.locals.success_msg = req.flash('success_msg');
        res.locals.error_msg = req.flash('error_msg');

        next();
});
...
```

Agregamos estos mensajes mediante el método **req.flash** a las rutas del archivo **/routes/deployments.js** en los métodos:

- **router.post** para la ruta **/deployments/newDeployment** para insertar un nuevo despliegue.

```
router.post('/deployments/newDeployment', (req, res) => {
        ...
        }else{
        const newDeployment = new Deployment({name, type, replicas
            });
        newDeployment.save();
        req.flash('success_msg', 'Despliegue Creado
            Satisfactoriamente');
```

```
        res.redirect('/deployments');
        }
});
```

- **router.put** para la ruta **/deployments/editDeployment/:id** para modificar un despliegue.

```
router.put('/deployments/editDeployment/:id', (req, res) => {
        Deployment.findByIdAndUpdate(req.params.id, req.body, (err
            , deploymentUpdated) => {
                if (err) return res.status(500).send({message: `
                    Error al actualizar el despliegue: ${err}`});
                if(!deploymentUpdated) return res.status(404).send
                    ({message: 'El despliegue no existe'});
                req.flash('success_msg', 'Despliegue Actualizado
                    Satisfactoriamente');
                res.redirect('/deployments');
        });
});
```

- **router.post** para la ruta **/deployments/newDeployment** para insertar un nuevo despliegue.

```
router.delete('/deployments/delete/:id', (req, res) => {
        ...
                deployment.remove(err => {
                        if (err) return res.status(500).send({
                            message: `Error al borrar el despliegue
                            : ${err}`});
                        req.flash('success_msg', 'Despliegue
                            Eliminado Satisfactoriamente');
                        res.redirect('/deployments');
                })
        })
})
```

Para que todas las vistas puedan mostrar estos mensajes, dentro de la carpeta **/views/partials** que almacenará archivos que podrán compartir todas las vistas de handlebars, creamos el archivo **messages.hbs** que recorrerá las variables flash globales de manera que si existe un mensaje de éxito, lo muestra como una alerta de bootstrap como ya vimos anteriormente.

```
{{#if success_msg}}
        <div class="alert alert-success alert-dismissible fade show"
            role="alert">
                {{success_msg}}
                <button type="button" class="close" data-dismiss="alert
                    " aria-label="Close">
                        <span aria-hidden="true">&times;</span>
                </button>
        </div>
{{/if}}
```

De la misma manera y por la misma razón que se ha realizado la implementación de los mensajes exitosos en el archivo **/views/partials/messages.hbs** en la parte superior, modificamos los mensajes de error introducidos en la parte superior del archivo **/views/deployments/newDeployment.hbs**, se eliminan de este archivo y se introducen en un nuevo archivo llamado **/views/partials/errors.hbs**.

Para utilizarlos vamos a la plantilla principal de las vistas de la aplicación situada en el archivo **/views/layouts/main.hbs** y la llamamos dentro del contenedor **container p-2** almacenado en el cuerpo, la manera de llamar a un archivo que pertenece a la carpeta partials es **{{{>nombreArchivoPartials }}}**.

```
...
<main class="container p-2">
        {{> messages}}
        {{> errors}}
        {{{ body }}}
</main>
...
```

De esta manera ya tenemos implementadas las funciones CRUD en el servidor junto a los mensajes de alertas flash con información referente a la acción realizada.

4.1.4. Implementación de funciones de navegación en el servidor

En esta apartado se implementará la navegación entre las diferentes funciones o rutas del servidor. Inicialmente comenzamos modificando el cuerpo de la plantilla principal de las vistas de la aplicación situada en el archivo **/view/layouts/-main.hbs**, cambiando el párrafo ** Navigation ** por la llamada a un archivo perteneciente a la carpeta partials que crearemos a continuación.

```
...
<body>

        {{> navigation }}
        ...
```

Creamos el archivo **/views/partials/navigation.hbs**, en el interior introduciremos un componente de navegación de bootstrap llamado **navbar**, esta navegación tendrá en su interior:

- Una pestaña **nav-item active** para ir a la sección de inicio implementada en la vista **/views/index.hbs**.
- Una pestaña **nav-item** para ir a la sección de test de balanceo de carga del servidor implementada en la vista **/views/test.hbs**.
- Una pestaña desplegable **nav-item dropdown** para ir a la sección de ver todos los despliegues almacenados en la base de datos implementada en la vista situada en el archivo **/views/deployments/allDeployments.hbs** como respuesta a una petición **GET** a la ruta **/deployments** situada en el archivo **/routes/deployments.js** o a la sección de insertar un nuevo despliegue en la base de datos implementada en la vista situada en el archivo **/views/deployments/newDeployment.hbs** como respuesta a una petición **GET** a la ruta **/deployments/add** situada en el archivo **/routes/deployments.js**.

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <div class="container">
                <a class="navbar-brand" href="/">KubeKVM App</a>
```

```
                <button class="navbar-toggler" type="button" data-
                    toggle="collapse" data-target="#navbarNav" aria-
                    controls="navbarNav" aria-expanded="false" aria-
                    label="Toggle navigation">
                        <span class="navbar-toggler-icon"></span>
                </button>
                <div class="collapse navbar-collapse" id="navbarNav">
                        <ul class="navbar-nav">
                                <li class="nav-item active">
                                        <a class="nav-link" href="/">
                                            Inicio <span class="sr-only"
                                            >(current)</span></a>
                                </li>
                                <li class="nav-item">
                                        <a class="nav-link" href="/test
                                            ">Test</a>
                                </li>
                                <li class="nav-item dropdown">
                                        <a class="nav-link dropdown-
                                            toggle" href="#" id="
                                            navbarDropdown" role="button
                                            " data-toggle="dropdown"
                                            aria-haspopup="true" aria-
                                            expanded="false">
                                                Despliegues
                                        </a>
                                        <div class="dropdown-menu" aria
                                            -labelledby="navbarDropdown"
                                            >
                                                <a class="dropdown-item
                                                    " href="/deployments
                                                    ">Ver Despliegues</a
                                                    >
                                                <a class="dropdown-item
                                                    " href="/deployments
                                                    /add">Crear Nuevo
                                                    Despliegue</a>
                                        </div>
                                </li>
```

```
                        </ul>
                    </div>
                </div>
</nav>
```

De esta manera ya tendremos desarrollada con total funcionalidad la API REST KubeKVM App a la cual realizarán peticiones los demás componentes del sistema final.

4.2. Script KVM

En esta sección se explica como se ha procedido al desarrollo del script KVM **main.py**, es decir se implementará un script en lenguaje **Python** que funcionando como un demonio consumirá los datos de los despliegues almacenados en al base de datos MongoDB a través de la API REST KubeKVM App y realizará la creación de las máquinas virtuales necesarias para cada despliegue utilizando la tecnología de virtualización KVM.

Uno de los primeros pasos es realizar la configuración inicial del sistema para implementar el script, dicha configuración se encuentra en el anexo con título **Anexo II: Configuración inicial para el desarrollo del script KVM**.

Tras haber realizado todos los pasos anteriores descritos en el anexo, ya disponemos del equipo configurado y listo para el desarrollo de esta API REST. Lo primero que debemos hacer el crear en el IDE el archivo llamado **main.py** que contendrá todas las acciones necesarias para obtener la funcionalidad deseada para este script.

Antes de comenzar debemos explicar los diferentes estados por los que pasan los despliegues almacenados en la base de datos, estos estados condicionan las diferentes funciones del script conformando una **máquina de estados finita** que funciona de la siguiente manera:

- Estado **inactivo**: en este estado se encuentra un despliegue almacenado en la base de datos, cuando ya han sido creadas todas las máquinas virtuales pertenecientes al mismo pero todas se encuentran apagadas. Puede ser modificado por el usuario a través de la aplicación realizando la acción de editar un despliegue que ya se encuentra almacenado en la base de datos, pasando el estado del despliegue de inactivo a **pendiente** que internamente utilizará la función **router.put('/deployments/editDeployment/:id')** situada en el archivo **/routes/deployments.js** perteneciente a la implementación de la API REST KubeKVM App.

- Estado **pendiente**: en este estado se encuentra un despliegue creado y almacenado en la base de datos por parte del usuario de la aplicación pero del cual aún no se han desplegado las máquinas virtuales para satisfacer las necesidades del mismo. Este estado es modificado por el script a través de la función **updateDeployment(url, state)** pasando el estado de pendiente a **activo** ,desplegando el número de máquinas virtuales necesarias para satisfacer las necesidades del mismo y encendiendo estas máquinas virtuales.

- Estado **activo**: en este estado se encuentra un despliegue almacenado en la base de datos, cuyas máquinas virtuales ya han sido desplegadas y se encuentran en funcionamiento. Este estado es modificado a través de la función **checkState(url, deployment)**:

  - Si aún siguen funcionando todas las máquinas virtuales pertenecientes a ese despliegue, el estado del despliegue permanece **activo**.
  - Si todas las máquinas virtuales pertenecientes a ese despliegue ya se encuentran apagadas, se pasa el estado del despliegue de activo a inactivo mediante la función **updateDeployment(url, state)**.

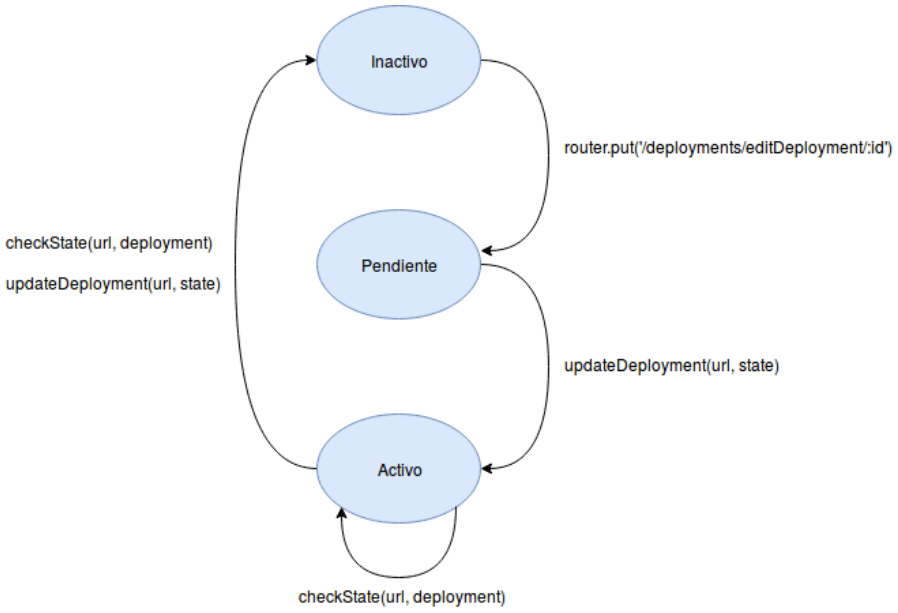


Figura 4.2: Máquina de estados finitos de un despliegue.

4.2.1. Implementación de la función principal main

Implementamos la función principal **main**, de manera que al ejecutar el script ejecuta el código dentro del condicional **if __name__ == '__main__':**, en su interior:

- Se almacena en una variable **url** la dirección de la aplicación a la que se enviará la petición GET para obtener los datos de los despliegues almacenados en la base de datos. Los despliegues están almacenados en la base de datos como se muestra en la imagen siguiente.

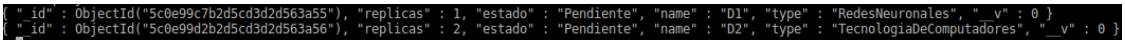


Figura 4.3: Formato de los datos almacenados en la base de datos.

- Se realiza la apertura de la conexión de la biblioteca de virtualización libvirt con el método **libvirt.open**. Para poder utilizar este método es necesario importar la librería **libvirt** al inicio del script.

```
import libvirt
```

- Se crea un diccionario vacío **deploymentsDictionary** para almacenar los datos de los despliegues almacenados en la base de datos retornados por la función **getDeployments**.
- Se realiza un bucle infinito **while True** para que haga la función de demonio:
  - Se almacena en el diccionario **deploymentsDictionary** todos los datos de los despliegues almacenados en la base de datos. Estos datos son retornados por la función **getDeployments** al que se le pasa como parámetro la **url** y el diccionario **deploymentsDictionary**. El diccionario devuelto por este método almacena como clave la id del despliegue y como valor una lista con los nombres de las máquinas virtuales desplegadas para satisfacer las necesidades de ese despliegue.

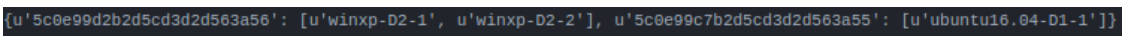


Figura 4.4: Elementos del diccionario tras 1 iteración del script.

- Se introduce un delay de tiempo mediante el método **time.sleep** para que el script se ejecute cada cierto periodo. Para poder utilizar esta método es necesario importar la librería **time** al inicio del script.

```
...
import time
...
if __name__ == '__main__':

        url='http://localhost:3000/deploymentsScript'
        conn=libvirt.open("qemu:///system")
        deploymentsDictionary={}

        while True:
                deploymentsDictionary=getDeployments(url,
                    deploymentsDictionary)
                time.sleep(30)
```

4.2.2. Implementación de la función getDeployments

Implementamos la función **getDeployments** que recibe como parámetros:

- La **url** con la dirección de la aplicación a la que se enviará la petición GET para obtener los datos de los despliegues almacenados en la base de datos.
- El diccionario **deploymentsDictionary** con todos los datos de los despliegues almacenados en la base de datos. Tras la primera iteración del script, este diccionario contiene un conjunto de pares clave:valor con la id del despliegue como clave y una lista con los nombres de las máquinas virtuales desplegadas para satisfacer las necesidades de ese despliegue como valor.

En el interior, mediante el método **requests.get** realizamos una petición GET a la **url** para obtener los datos de los despliegues almacenados en la base de datos y lo guardamos en la variable **response**. Para poder utilizar este método es necesario importar la librería **requests** al inicio del script.

```
...
import requests
```

```
...
```

- Si la petición se realiza correctamente **if response.status_code==200**, para tener un mejor manejo de los datos, almacenamos en la variable **payload** un diccionario en formato JSON que obtenemos utilizando el método **response.json**. Ahora mediante el método **payload.get** buscamos los despliegues en el diccionario pasándole la cadena **deployments**, junto a ella se envía una lista vacía para que almacene en cada posición un diccionario con datos de cada despliegue y retorne la lista a una variable llamada **deployments**.
    - Si la lista almacenada en **deployments** no está vacía **if deployments**, se recorren los diccionarios de despliegues de la misma.
        - Si el valor correspondiente a la clave estado del diccionario es Pendiente **if deployment['estado']=='Pendiente'**, se inserta en el diccionario **deploymentsDictionary** el valor de la id del despliegue como clave y una lista vacía en la que a continuación se guardarán los nombres de las máquinas virtuales desplegadas para satisfacer las necesidades de ese despliegue como valor. **deploymentsDictionary[deployment['_id']]=[]**. Se realiza la actualización del estado del despliegue utilizando la función **updateDeployment** que se implementará más adelante, a la que se le pasa la url de la ruta de la aplicación correspondiente a la actualización de un despliegue concatenada con la id del despliegue **http://localhost:3000/deployments/editDeployment/'+deployment['_id']** y el estado al que se actualizará dicho despliegue **Activo**. Tras esto se realiza el despliegue de las máquinas virtuales necesarias para satisfacer los requisitos del despliegue utilizando la función **kvmDeployment** que se implementará más adelante, al que se le pasa el tipo del despliegue a realizar **deployment['type']**, el número de replicas que deben crearse **deployment['replicas']** y el nombre del despliegue **deployment['name']**. Este método introduce en el diccionario **deploymentsDictionary** en la posición que tiene la id del despliegue como clave, una lista con los nombres de las máquinas virtuales desplegadas para sa-

tisfacer las necesidades de ese despliegue como valor. Para finalizar se encienden las máquinas virtuales creadas para ese despliegue ejecutando la función **startDeploymentMachines** pasándole la lista con los nombres de esas máquinas virtuales **deploymentsDictionary[deployment['_id']]**. Al salir, la función **getDeployments** devuelve el diccionario **deploymentsDictionary** en el que están almacenados un conjunto de pares clave:valor correspondientes a todos los despliegues almacenados en la base de datos con el valor de la id del despliegue como clave y una lista con los nombres de las máquinas virtuales desplegadas para satisfacer las necesidades de ese despliegue como valor.

- Si el valor correspondiente a la clave estado del diccionario es Activo **if deployment['estado']=='Activo'** se recorre el diccionario **deploymentsDictionary**, si la id del diccionario es la de ese despliegue **if deploymentsDictionary.keys()[i]==deployment['_id']** se ejecuta la función **checkState** que comprueba el estado de las máquinas virtuales para ver si están apagadas y puede modificarse su estado a Inactivo. A esta función se le pasa la url de la ruta de la aplicación correspondiente a la actualización de un despliegue **http://localhost:3000/deployments/editDeployment/** y una tupla que contiene el id y el nombre de las máquinas de ese despliegue **deploymentsDictionary.items()[i]**. Al salir la función **getDeployments** devuelve el diccionario **deploymentsDictionary** en el que están almacenados un conjunto de pares clave:valor correspondientes a todos los despliegues almacenados en la base de datos con el valor de la id del despliegue como clave y una lista con los nombres de las máquinas virtuales desplegadas para satisfacer las necesidades de ese despliegue como valor.
- Si el valor correspondiente a la clave estado del diccionario no es ni Pendiente ni activo, devuelve el diccionario **deploymentsDictionary** sin modificar.

- Si la lista almacenada en **deployments** está vacía, devuelve el diccionario **deploymentsDictionary** sin modificar.

- Si la petición no se realiza correctamente, devuelve el diccionario **deploymentsDictionary** sin modificar.

```
def getDeployments(url, deploymentsDictionary):

        response = requests.get(url)
        if response.status_code==200:
                payload=response.json()
                deployments=payload.get('deployments',[])
                if deployments:#Si no esta vacia
                        for deployment in deployments:
                                if deployment['estado']=='Pendiente':
                                        deploymentsDictionary[
                                            deployment['_id']]=[]
                                        updateDeployment('http://
                                            localhost:3000/deployments/
                                            editDeployment/'+deployment[
                                            '_id'], 'Activo')
                                        deploymentsDictionary[
                                            deployment['_id']]=
                                            kvmDeployment(deployment['
                                            type'], deployment['replicas
                                            '], deployment['name'])
                                        startDeploymentMachines(
                                            deploymentsDictionary[
                                            deployment['_id']])
                                elif deployment['estado']=='Activo':
                                        for i in range(len(
                                            deploymentsDictionary)):
                                                if
                                                    deploymentsDictionary
                                                    .keys()[i]==
                                                    deployment['_id']:
                                                        checkState('
                                                            http://
                                                            localhost
```

```
                                                                    :3000/
                                                                    deployments/
                                                                    editDeployment
                                                                    /',
                                                                    deploymentsDictionary
                                                                    .items()[i])
        return deploymentsDictionary
```

4.2.3. Implementación de la función updateDeployment

Implementamos la función **updateDeployments** que recibe como parámetros:

- La **url** con la dirección de la aplicación concatenada con la id del despliegue a la que se enviará la petición PUT para actualizar los datos del mismo.
- El estado **state** al que se actualizará dicho despliegue.

En el interior almacenamos en un diccionario llamado **payload** un par clave:valor con la clave **estado** y el valor **state**, que es el valor recibido por parámetro. A continuación, mediante el método **requests.get** realizamos una petición GET a la **url** pasándole el diccionario payload como atributo **data**, de esta manera actualizamos los datos del despliegue y le respuesta a la petición la guardamos en la variable **response**. Si la petición se realiza correctamente **if response.status_code==200**, se muestra por el terminal un mensaje de texto para informar de la modificación satisfactoria del despliegue al nuevo estado.

```
def updateDeployment(url, state):

        payload={'estado':state}
        response=requests.put(url, data=payload)
        if response.status_code==200:
                print('Despliegue actualizado satisfactoriamente al
                    estado '+state)
```

4.2.4. Implementación de la función startDeploymentMachines

Implementamos la función **startDeploymentMachines** que recibe como parámetro la lista **deploymentsMachines** con los nombres pertenecientes a las máquinas virtuales creadas para un despliegue. Se recorre la lista recibida por parámetro y por cada nombre almacenado en esta lista se almacena en la variable **command** una cadena de texto que contiene el comando de terminal necesario para iniciar una máquina virtual KVM **virsh start** concatenada con el nombre de la misma. Utilizando la función **subprocess.Popen** se obtiene en tiempo real la salida por terminal. Para poder utilizar esta método es necesario importar las librerías **sys** y **subprocess** al inicio del script.

```
...
import sys, subprocess
...
def startDeploymentMachines(deploymentsMachines):

        for name in deploymentsMachines:
                command = 'sudo virsh start '+name
                result = subprocess.Popen(command, shell=True, stdout=
                    subprocess.PIPE)
                for output in result.stdout:
                        print(output.decode(sys.getdefaultencoding()).
                            rstrip())
```

4.2.5. Implementación de la función kvmDeployment

Implementamos la función **kvmDeployments** que recibe como parámetros el tipo de despliegue a realizar **type**, el número de replicas que deben crearse **replicas** y el nombre del despliegue **name**. Se crea la lista **machineNames** para almacenar los nombres de las máquinas virtuales creadas. Las siguientes operaciones se realizan tantas veces como el número de replicas recibidas por parámetro:

- Si el tipo de despliegue recibido por parámetro es el de la asignatura de Tecnologia de Computadores **if type=='TecnologiaDeComputadores':**, se inserta al final de la lista **machineNames** el nombre de la máquina virtual **winxp** concatenado con el nombre del despliegue recibido por parámetro **name** y a la vez concatenado con un iterador **i** para identificar esa máquina virtual. A continuación, se almacena en la variable **command** una cadena de texto que contiene el comando de terminal necesario para clonar una máquina virtual KVM **virt-clone –connect qemu:///system –original winxp –name winxp-'+name+'-'+str(i+1)+' –file /var/lib/libvirt/images/winxp-'+name+'-'+str(i+1)+'.qcow2** a partir de las imágenes básicas para cada tipo de despliegue creadas en **Anexo II: Configuración inicial para el desarrollo del script KVM**. Para terminar, utilizando la función **subprocess.Popen** se obtiene en tiempo real la salida por terminal.

- Si el tipo de despliegue recibido por parámetro es el de la asignatura de Redes Neuronales **if type=='RedesNeuronales':**, se inserta al final de la lista **machineNames** el nombre de la máquina virtual **ubuntu16.04** concatenado con el nombre del despliegue recibido por parámetro **name** y a la vez concatenado con un iterador **i** para identificar esa máquina virtual. A continuación, se almacena en la variable **command** una cadena de texto que contiene el comando de terminal necesario para clonar una máquina virtual KVM **virt-clone –connect qemu:///system –original ubuntu16.04 –name ubuntu16.04-'+name+'-'+str(i+1)+' –file /var/lib/libvirt/images/ubuntu16.04-'+name+'-'+str(i+1)+'.qcow2** a partir de las imágenes básicas para cada tipo de despliegue creadas en **Anexo II: Configuración inicial para el desarrollo del script KVM**. Para terminar, utilizando la función **subprocess.Popen** se obtiene en tiempo real la salida por terminal.

Esta función retorna la lista **machineNames** con los nombres de las máquinas virtuales desplegadas para satisfacer las necesidades de ese despliegue.

```
def kvmDeployment(type, replicas, name):

        machineNames=[]
        for i in range(replicas):
                if type=='TecnologiaDeComputadores':
                        machineNames.append('winxp-'+name+'-'+str(i+1))
                        command = 'sudo virt-clone --connect qemu:///
                            system --original winxp --name winxp-'+name+
                            '-'+str(i+1)+' --file /var/lib/libvirt/
                            images/winxp-'+name+'-'+str(i+1)+'.qcow2'
                        result = subprocess.Popen(command, shell=True,
                            stdout=subprocess.PIPE)
                        for output in result.stdout:
                                print(output.decode(sys.
                                    getdefaultencoding()).rstrip())
                elif type=='RedesNeuronales':
                        machineNames.append('ubuntu16.04-'+name+'-'+str
                            (i+1))
                        command = 'sudo virt-clone --connect qemu:///
                            system --original ubuntu16.04 --name
                            ubuntu16.04-'+name+'-'+str(i+1)+' --file /
                            var/lib/libvirt/images/ubuntu16.04-'+name+'-
                            '+str(i+1)+'.qcow2'
                        result = subprocess.Popen(command, shell=True,
                            stdout=subprocess.PIPE)
                        for output in result.stdout:
                                print(output.decode(sys.
                                    getdefaultencoding()).rstrip())
        return machineNames
```

4.2.6. Implementación de la función checkState

Implementamos la función **checkState** que recibe como parámetros la url de la ruta de la aplicación correspondiente a la actualización de un despliegue **url** y una tupla que contiene el id del despliegue junto con el nombre de las máquinas de ese despliegue **deployment**. Se recorre la segunda posición de la tupla recibida **deployment[1]** que es donde están almacenados los nombres de las máquinas virtuales, por cada nombre de en la lista de nombres de máquinas virtuales se recorren los nombres de los dominios relacionados con máquinas virtuales existentes en KVM mediante el método **conn.listAllDomains**. Si el nombre almacenado en la lista está entre los nombres de los dominios KVM **if name in dom.name():** y si el estado de ese dominio se encuentra inactivo, es decir la máquina virtual esta apagada **if dom.state()[0]==5:**, se aumenta un iterador que llevará la cuenta del número de máquinas virtuales inactivas que corresponden a ese despliegue. Si todas las máquinas virtuales del despliegue están apagadas **if i==len(deployment[1]):**, Se actualiza el estado del despliegue a Inactivo mediante la función **updateDeployment** a la que se le pasa la url de la ruta de la aplicación correspondiente a la actualización de un despliegue recibida por parámetro **url** concatenada con la id del despliegue almacenada en la primera posición de la tupla recibida por parámetro **deployment[0]** y el estado al que se actualizará dicho despliegue **Activo**.

```
def checkState(url, deployment):

        i=0
        for name in deployment[1]:
                for dom in conn.listAllDomains():
                        if name in dom.name():
                                if dom.state()[0]==5:
                                        i+=1
                                        if i==len(deployment[1]):
                                                updateDeployment(url+
                                                    deployment[0], '
                                                    Inactivo')
```

De esta manera ya tendremos desarrollada con total funcionalidad el script KVM **main.py** en lenguaje **Python** que funcionando como un demonio consumirá los datos de los despliegues almacenados en al base de datos MongoDB a través de la API REST KubeKVM App y realizará la creación de las máquinas virtuales necesarias para cada despliegue utilizando la tecnología de virtualización KVM.

4.3. Docker y Kubernetes

En esta sección se explica como se ha procedido al desarrollo de los archivos **Dockerfile** de **Docker** y **yaml** de **Kubernetes** para realizar la exposición de los servicios ofrecidos por la **API REST KubeKVM App** a través de **Kubernetes**.

Uno de los primeros pasos es realizar la configuración inicial del sistema para implementar estos archivos, dicha configuración se encuentra en el anexo con título **Anexo III: Configuración inicial para la creación del cluster de Kubernetes**.

Tras haber realizado todos los pasos anteriores descritos en el anexo, ya disponemos del equipo configurado y listo para el desarrollo de estos archivos.

4.3.1. Implementación del archivo Dockerfile

Dentro de la carpeta en la que se ha desarrollado la aplicación **KubeKVM App**, implementamos el archivo **Dockerfile**, que contiene una serie de instrucciones para automatizar el proceso de creación de un contenedor **Docker**.

En el interior de este archivo se ejecutarán los siguientes comandos **Dockerfile**:

- **FROM**: se indica la imagen base a partir de la cual se construirá el contenedor junto a un tag, es este caso el tag **latest** le dice a **Docker** que busque la última versión de esta imagen. De esta manera, Docker buscar en la máquina local una imagen con ese nombre y si no la encuentra, la descarga de los repositorios. En este caso le decimos que utilice una imagen de **node**.
- **RUN**: ejecuta comandos dentro del contenedor, aplica estos comandos creando encima una nueva capa con los cambios producidos por la ejecución del comando y sigue con la siguiente instrucción. En este caso le decimos que cree la carpeta junto a los directorios padre que falten para cada argumento directorio **/usr/src/app** en donde se almacenará la aplicación **KubeKVM App**.

- **WORKDIR**:permita configurar sobre que directorio se ejecutara una instrucción RUN, CMD o ENTRYPOINT. En este caso se refiere al siguiente comando **RUN** introducido en el archivo que se ejecutará sobre el directorio **/usr/src/app**.

- **COPY**: realiza la copia de los archivos o directorios de una ubicación especificada como fuente y los agrega al sistema de archivos del contenedor en la ruta especificada como destino. En este caso realiza la copia del archivo **package.json** que describe todo el proyecto en la carpeta **/usr/src/app** donde se almacenará la aplicación.

- **RUN**: instalar el gestor de paquetes **npm**, el cual permitirá instalar fácilmente módulos y paquetes para usar con Node.js en el directorio **/usr/src/app** especificado en la parte superior a través del comando **WORKDIR**.

- **COPY**: realiza la copia de los archivos y directorios de la carpeta actual, que es la carpeta en la que se encuentra la aplicación **KubeKVM App** en el directorio **/usr/src/app**.

- **EXPOSE**: realiza la asociación de puertos, permitiendo exponer un contenedor al exterior. De esta manera, Docker sabe que el contenedor escucha en los puertos especificados. En este caso el contenedor escucha por el puerto **3000** que es el puerto que se configura para que escuche la aplicación **KubeKVM App**.

- **ENV**: realiza el establecimiento de variables de entorno para el contenedor. En este caso se establecerán las variables **MONGODB** y **PORT** que serán leídas por el archivo de configuración **config.js** de la aplicación **KubeKVM App**.

- **CMD**: realiza la misma acción que comando RUN pero con la diferencia de que se ejecuta cuando se instancia o se arranca el contenedor. Por cada archivo Dockerfile solo puede haber un comando **CMD** y es de gran utilidad para ejecutar servicios instalados o archivos ejecutables. En este caso se ejecuta el comando **npm start** que lanzará la aplicación **KubeKVM App** dentro del contenedor.

```
FROM node:latest
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY package.json /usr/src/app/
RUN npm install

COPY . /usr/src/app
EXPOSE 3000
ENV MONGODB = mongodb://mongo:27017/deploymentsKVM
ENV PORT=3000
CMD ["npm", "start"]
```

Tras esto, en la misma ubicación creamos el archivo **.dockerignore**, que servirá para que el archivo **Dockerfile** ignore los archivos o directorios introducidos dentro de este.

```
node_modules
npm-debug.log
```

4.3.2. Implementación de los archivos YAML

Dentro de la carpeta en la que se ha desarrollado la aplicación **KubeKVM App**, creamos la carpeta **kubernetes**. En su interior, implementamos los archivos **api.yaml** y **mongo.yaml**, que contienen una serie de instrucciones para la creación de los componentes de **Kubernetes** necesarios para el despliegue de la aplicación **KubeKVM App**.

Estos archivos contienen en el interior las siguientes definiciones:

- **apiVersion**: número de versión del api que se quiere utilizar.
- **kind**: tipo de fichero que se va a crear.
- **metadata**: datos propios del tipo como el **nombre** y los **labels** tipo clave:valor que tiene asociados para seleccionarlo.
- **spec**: contiene la especificación del tipo.

- **containers**: se nombran los contenedores que forman parte de ese tipo especificando en su interior el **nombre**, la **imagen** que utiliza y el **puerto** por el que escucha ese contenedor. Todos estos contenedores serían visibles por localhost.
- **replicas**: número de réplicas que queremos que se encargue de mantener el controlador de replicación.
- **selector**: se indican todos los pods que se va a encargar de gestionar el servicio. Estos datos son **labels** de tipo clave:valor.
- **template**: tiene exactamente el mismo esquema interno que un pod , excepto que como está anidado no necesita ni un apiVersion ni un kind, es decir contiene en su interior **metadata** y **spec**.
- **ports**: en el caso de ser un servicio, se indican los puertos utilizados para exponer el mismo.
- **type**: en el caso de ser un servicio, se indica el tipo de acceso al mismo.

Se implementa el archivo **api.yaml** correspondiente a la aplicación **KubeKVM App**.

```
apiVersion: v1
kind: Service
metadata:
        name: nodeapp
        labels:
                run: nodeapp
spec:
        ports:
        - port: 80
                targetPort: 3000
                protocol: TCP
                nodePort: 81
        type: NodePort
        selector:
                run: nodeapp


---
apiVersion: extensions/v1beta1
```

```
kind: Deployment
metadata:
        name: nodeapp
spec:
        replicas: 1
        template:
                metadata:
                        labels:
                                run: nodeapp
                spec:
                        containers:
                                - name: myapi
                                image: index.docker.io/pmoralof/
                                    appvisual:latest
                                ports:
                                - containerPort: 3000
```

Se implementa el archivo **mongo.yaml** correspondiente a la base de datos **MongoDB** relacionada con la aplicación **KubeKVM App**.

```
apiVersion: v1
kind: Service
metadata:
        name: mongo
        labels:
                run: mongo
spec:
        ports:
        - port: 27017
                targetPort: 27017
                protocol: TCP
        selector:
                run: mongo
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
        name: mongo
spec:
        template:
```

```
    metadata:
            labels:
                    run: mongo
    spec:
            containers:
            - name: mongo
                    image: mongo
                    ports:
                    - containerPort: 27017
```

De esta manera, ya tenemos desarrollados los archivos **Dockerfile** de **Docker** y **yaml** de **Kubernetes** para realizar la exposición de los servicios ofrecidos por la **API REST KubeKVM App** a través de **Kubernetes**.

Capítulo 5

Manual del usuario

En este apartado se expondrán las diferentes acciones que puede realizar el usuario de la aplicación desarrollada. Antes de realizar la conexión con el servicio a través del navegador, debe conocerse la **dirección** del mismo y el **puerto**.

Tras acceder a la dirección del servicio, se le proporciona al cliente la **página principal** de la aplicación web en la que se presenta una breve descripción de la misma y un enlace hacia la página web de la Universidad de Extremadura. A través de las diferentes pestañas situadas en la parte superior puede accederse a las diferentes secciones disponibles en la página web. Podemos observar la estructura.

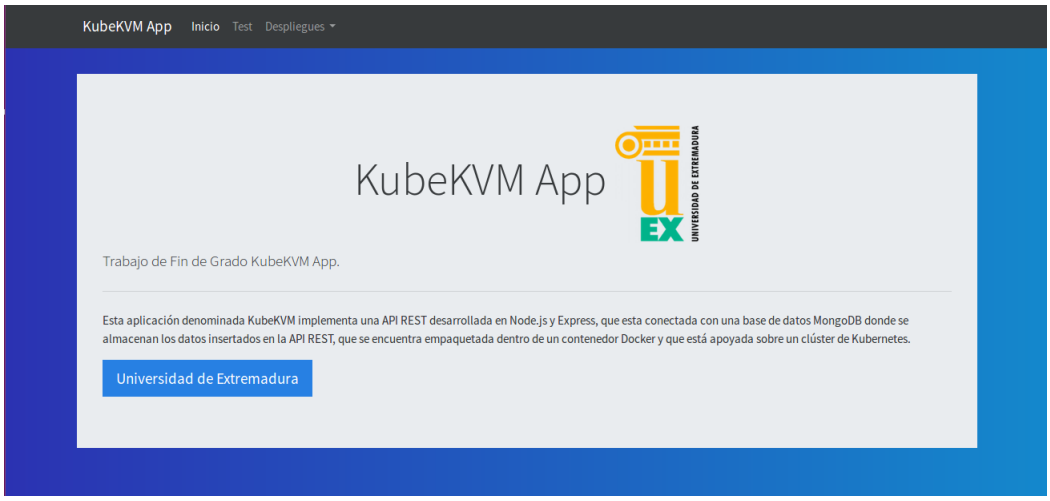


Figura 5.1: Página principal de la aplicación web.

En la pestaña **test** puede consultarse la información referente al **pod** en el que se está ejecutando la aplicación, ya que donde se ejecute la misma depende del balanceo de carga realizado en el servidor por el cluster de Kubernetes sobre el que esta apoyada la aplicación.

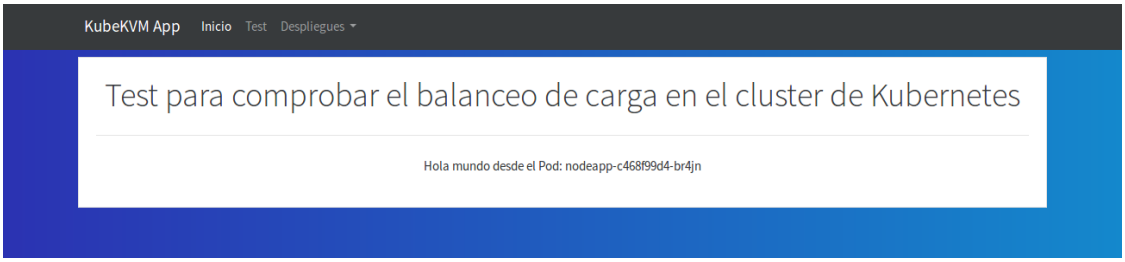


Figura 5.2: Test de balanceo de carga de la aplicación web.

En la pestaña **despliegues** se obtiene un desplegable para realizar acciones relativas a los despliegues.

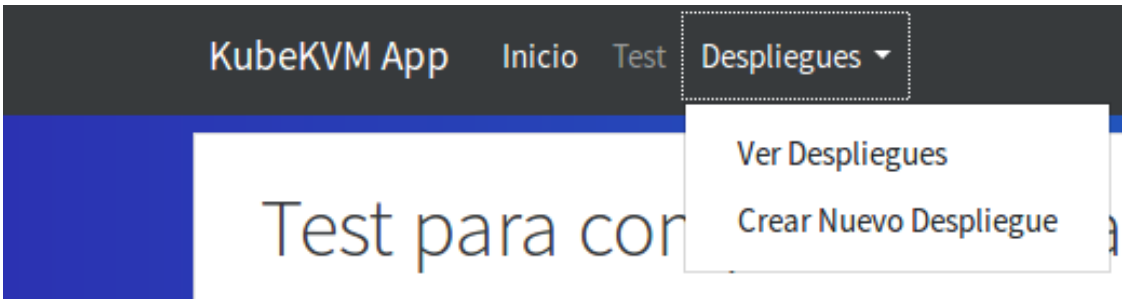


Figura 5.3: Desplegable de los despliegues de la aplicación web.

En el interior de la pestaña **despliegues**, si se selecciona la opción **crear nuevo despliegue** se obtiene un formulario para proceder a la creación de un nuevo despliegue. En el interior del mismo puede especificarse el **nombre**, **tipo** y **número de replicas** del despliegue.

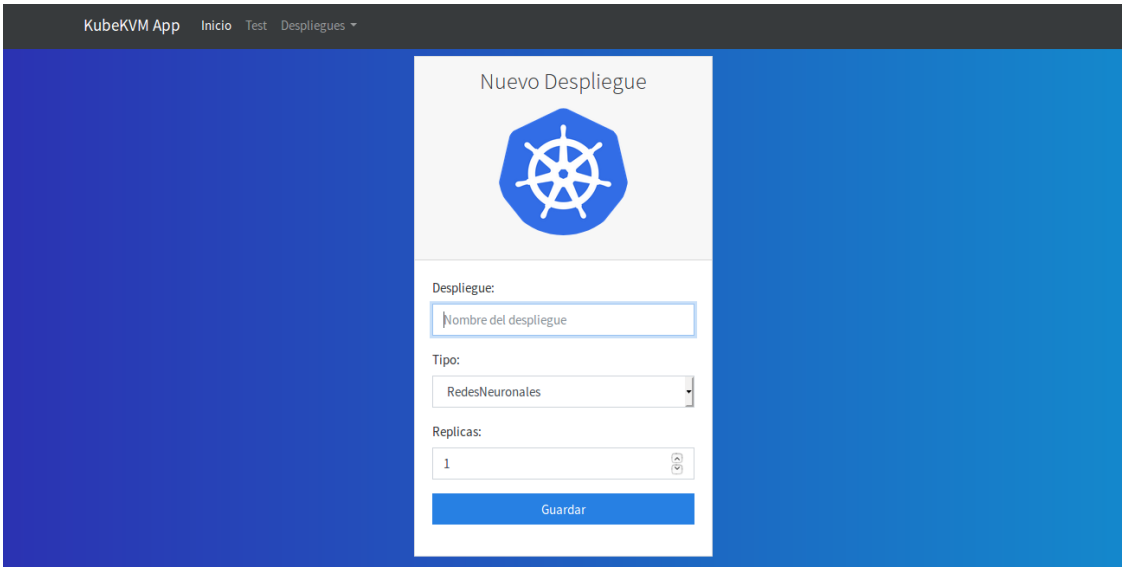


Figura 5.4: Formulario para la creación de un nuevo despliegue de la aplicación web.

En el interior de la pestaña **despliegues**, si se selecciona la opción **ver despliegues** se puede consultar información relativa a los despliegues almacenados en la base de datos. Si no hay ningún despliegue almacenado en la base de datos, en la ventana se mostrará un botón que direcciona hacia el formulario para la creación de un nuevo despliegue.

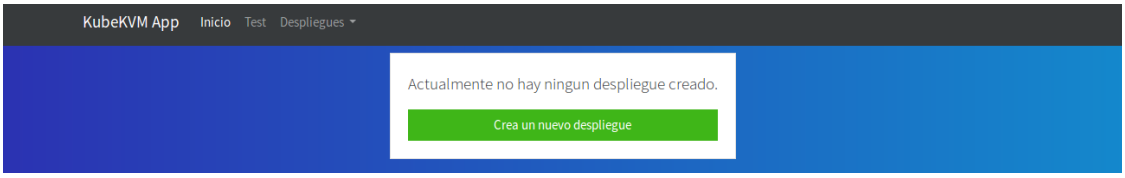


Figura 5.5: Ventana ver despliegues de la aplicación web sin ningún despliegue introducido.

Si hay despliegues almacenados en la base de datos, se mostrarán en la ventana con 2 botones para cada despliegue creado que hacen referencia a **editar** o **eliminar** ese despliegue.

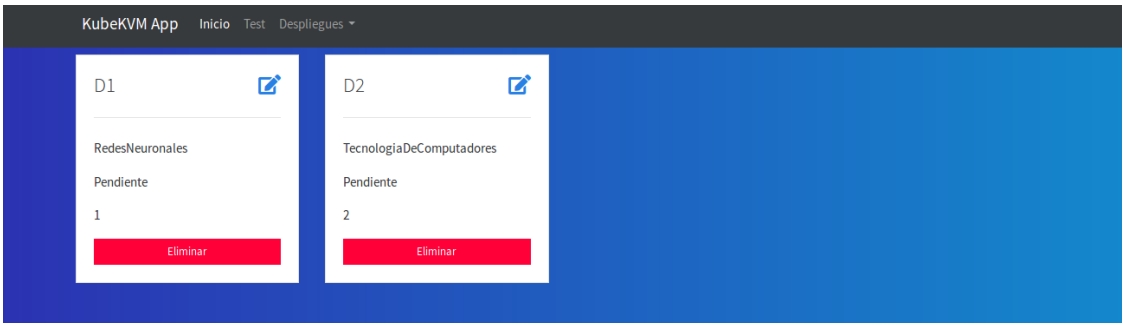


Figura 5.6: Ventana ver despliegues de la aplicación con despliegues introducidos.

Se se selecciona el botón para **editar despliegue** se obtiene un formulario para proceder a la edición del despliegue. En el interior puede modificarse el **nombre**, **tipo**, **estado** y **número de replicas** del despliegue.

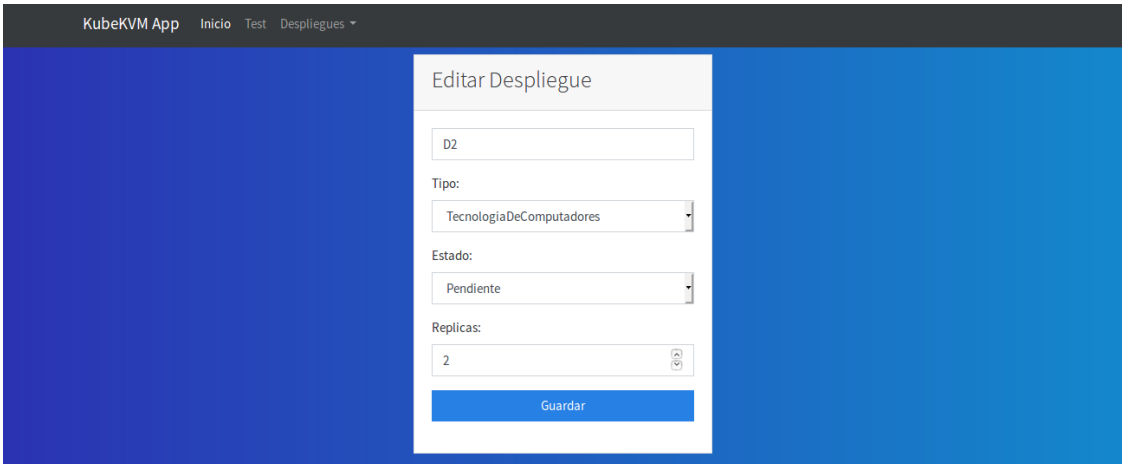


Figura 5.7: Formulario para la edición de un despliegue de la aplicación web.

Capítulo 6

Manual del desarrollador

Para poder poner en total funcionamiento el proyecto y permitir en un futuro el desarrollo de una mejora o simplemente ver su funcionamiento interno, ademas de seguir los pasos de configuración detallados en los anexos **Anexo II: Configuración inicial para el desarrollo del script KVM** y **Anexo III: Configuración inicial para la creación del cluster de Kubernetes**, es necesario realizar una serie de configuraciones.

6.1. Creación de la imagen Docker

La imagen **Docker** correspondiente a la aplicación consumida por el archivo **api.yaml** debe ser creada previemente. Para esto, inicialmente nos situamos en la carpeta que contiene la aplicación construimos la imagen a partir del fichero **Dockerfile** ejecutando:

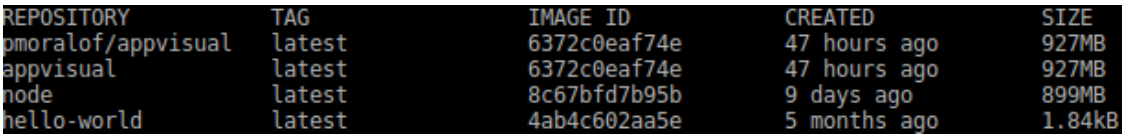
```
sudo docker build -t appvisual .
```

Procedemos a etiquetar la imagen:

```
sudo docker tag appvisual:latest pmoralof/appvisual:latest
```

Realizamos una comprobación de las imágenes creadas mediante:

```
sudo docker images
```



REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
pmoralof/appvisual	latest	6372c0eaf74e	47 hours ago	927MB
appvisual	latest	6372c0eaf74e	47 hours ago	927MB
node	latest	8c67bfd7b95b	9 days ago	899MB
hello-world	latest	4ab4c602aa5e	5 months ago	1.84kB

Figura 6.1: Imagenes docker creadas.

Para finalizar, subimos la imagen creada al repositorio **Docker Hub** para que pueda ser descargada por el archivo **api.yaml** cuando se proceda a su despliegue en **Kubernetes**:

```
sudo docker push pmoralof/appvisual:latest
```

6.2. Despliegue de la aplicación en Kubernetes

Una vez creado e iniciado el cluster de **Kubernetes**, se realiza del despliegue de la aplicación en su interior mediante el uso de los archivos **yaml** implementados anteriormente.Para comenzar copiamos en el nodo **master** los archivos:

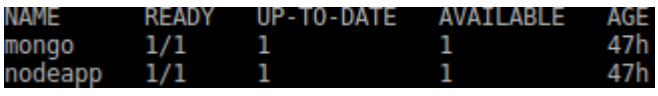
```
scp -r /home/pablo/Escritorio/TFG/AppVisual/kubernetes/api.yaml
    paco@158.49.112.74:/home/paco
scp -r /home/pablo/Escritorio/TFG/AppVisual/kubernetes/mongo.yaml
    paco@158.49.112.74:/home/paco
```

Procedemos al despliegue de la aplicación. Una vez conectados a través de **ssh** al nodo **master**, ejecutamos:

```
kubectl apply -f mongo.yaml
kubectl apply -f api.yaml
```

Comprobamos que están funcionando los **despliegues** definidos:

```
kubectl get deployments
```



NAME	READY	UP-TO-DATE	AVAILABLE	AGE
mongo	1/1	1	1	47h
nodeapp	1/1	1	1	47h

Figura 6.2: Despliegues Kubernetes.

Comprobamos los **pods** definidos que se están ejecutando:

```
kubectl get pods
```

```
NAME                       READY   STATUS    RESTARTS   AGE
mongo-6456979955-8ft7d     1/1     Running   0          47h
nodeapp-6864f57974-gz9wj   1/1     Running   0          47h
```

Figura 6.3: Pods Kubernetes.

Para finalizar, realizamos la última comprobación para ver que los **servicios** definidos se ofertan correctamente:

```
kubectl get services
```

```
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP          6d3h
mongo        ClusterIP   10.98.89.155   <none>        27017/TCP        47h
nodeapp      NodePort    10.110.97.67   <none>        80:30885/TCP     47h
```

Figura 6.4: Servicios Kubernetes.

6.3. Despliegue del scriptKVM

Para poner a funcionar como un servicio el script KVM **main.py** implementado anteriormente, se necesitan realizar un conjunto de configuraciones en el nodo **master**. Inicialmente iniciamos la red por defecto:

```
sudo virsh net-start default
```

Configuramos la red por defecto para que se inicie siempre de forma automática:

```
sudo virsh net-autostart default
```

Listamos la configuración de red disponible:

```
sudo virsh net-list
```

```
Nombre               Estado     Inicio automático Persistente
----------------------------------------------------------
default              activo     si                si
```

Figura 6.5: Configuración de red disponible.

Realizamos la copia de las imágenes de las máquinas virtuales básicas **winxp.qcow2** y **ubuntu16.04.qcow2** al directorio **/var/lib/libvirt/images/** y de las configuraciones de las máquinas virtuales básicas **winxp.xml** y **ubuntu16.04.xml** al directorio **/etc/libvirt/qemu/**. A continuación, reiniciamos el sistema **libvirt**:

```
systemctl restart libvirtd
```

Comprobamos que se ven los dominios referentes a las máquinas virtuales:

```
virsh list --all
```

```
Id    Nombre                         Estado
----------------------------------------------------
-     ubuntu16.04                    apagado
-     winxp                          apagado
```

Figura 6.6: Dominios de las máquinas virtuales.

Modificamos el tipo de procesador definido en los archivos **xml** referentes a las configuraciones de las máquinas virtuales básicas:

- Vemos el tipo de procesador del nodo master:

```
virsh capabilities | less
```

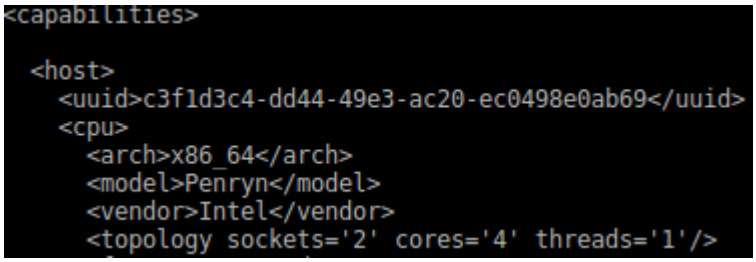


Figura 6.7: Tipo de procesador del nodo master.

- Editamos el tipo de procesador de las configuraciones de las máquinas virtuales básicas:

```
virsh edit winxp
virsh edit ubuntu16.04
```

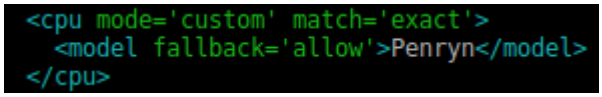


Figura 6.8: Modificación de la configuración de las VM básicas.

Copiamos el **scriptKVM** al directorio **/lib/systemd/system/** e implementamos el servicio **kvm.service** para este:

```
[Unit]
Description=Servicio del script KVM

[Service]
Type=simple
ExecStart=/usr/bin/python2.7 /lib/systemd/system/main.py
Restart=on-abort
```

```
[Install]
WantedBy=multi-user.target
```

Modificamos el script KVM **main.py** cambiando en las **url** la dirección **localhost** y el puerto **3000** por la dirección expuesta por el servicio y el puerto **80** del mismo.

```
nodeapp      NodePort    10.110.97.67   <none>        80:30885/TCP   2d23h
```

Figura 6.9: Dirección y puertos expuestos por el servicio de la aplicación.

Asignamos los permisos necesarios al servicio **kvm.service** y al script KVM **main.py**:

```
sudo chmod 644 kvm.service
sudo chmod 644 main.py
sudo chmod +x kvm.service
```

Recargamos los demonios:

```
sudo systemctl daemon-reload
```

Habilitamos el servicio **kvm.service** y lo iniciamos:

```
systemctl enable kvm.service
systemctl start kvm.service
```

Comprobamos que el servicio **kvm.service** esta funcionando correctamente:

```
systemctl status kvm
```

```
● kvm.service - Servicio del script KVM
   Loaded: loaded (/lib/systemd/system/kvm.service; enabled; vendor preset: enabled)
   Active: active (running) since lun 2019-02-04 18:05:01 CET; 1 day 3h ago
 Main PID: 315 (python2.7)
    Tasks: 1
   Memory: 21.6M
      CPU: 12.567s
   CGroup: /system.slice/kvm.service
           └─315 /usr/bin/python2.7 /lib/systemd/system/main.py

feb 05 21:25:39 master python2.7[315]: ------------------
feb 05 21:25:39 master python2.7[315]: {u'5c56044da0be780011360d6b': [u'ubuntu16.04-D2-1'], u'5c560435a0be780011360d6a': [u'winxp-D1-1']}
feb 05 21:25:39 master python2.7[315]: {u'name': u'D1', u'replicas': 1, u'estado': u'Inactivo', u'__v': 0, u'_id': u'5c560435a0be780011360d6a', u'type': u'Tecnologi
feb 05 21:25:39 master python2.7[315]: {u'name': u'D2', u'replicas': 1, u'estado': u'Inactivo', u'__v': 0, u'_id': u'5c56044da0be780011360d6b', u'type': u'RedesNeur
feb 05 21:25:39 master python2.7[315]: ======================================================================
feb 05 21:25:39 master python2.7[315]: ======================================================================
feb 05 21:25:39 master python2.7[315]: DICCIONARIO EN GET
feb 05 21:25:39 master python2.7[315]: ------------------
feb 05 21:25:39 master python2.7[315]: {u'5c56044da0be780011360d6b': [u'ubuntu16.04-D2-1'], u'5c560435a0be780011360d6a': [u'winxp-D1-1']}
feb 05 21:25:39 master python2.7[315]: {u'name': u'D1', u'replicas': 1, u'estado': u'Inactivo', u'__v': 0, u'_id': u'5c560435a0be780011360d6a', u'type': u'Tecnologi
```

Figura 6.10: Servicio kvm.service.

De esta manera, el proyecto ya se encuentra en total funcionamiento y permite posibles desarrollos de mejoras o que simplemente pueda observarse su funcionamiento interno.

Capítulo 7

Conclusiones

Tras finalizar el desarrollo de la aplicación final **KubeKVM**, podemos llegar a la conclusión que aparte del proporcionar un servicio rápido, eficiente y con alta disponibilidad, ofrece un alto grado de flexibilidad, portabilidad, automatización, autoregeneración ante fallos, y una gran cantidad de ventajas que provienen del encapsulado en **Docker** y del cluster de **Kubernetes** sobre el que se encuentra apoyado.

Además de las conclusiones anteriores, el despliegue de este servicio es simplificado en gran manera por el uso de los archivos **yaml** utilizados por Kubernetes, ya que definiendo estos archivos para satisfacer las necesidades del servicio y del despliegue a realizar, se obtiene la **API REST** que oferta la aplicación final con solo ejecutar 2 órdenes **kubectl** para la creación del mismo.

En el ámbito de la **Escuela Politécnica de Cáceres**, se ofrece a los docentes una plataforma para facilitar la preparación y el desempeño de los laboratorios de las asignaturas impartidas por los mismos, destacando las ventajas de escalabilidad y reusabilidad para las posibles necesidades que puedan surgir.

Como parte final, el desarrollo realizado en el proyecto servirá de base para futuras mejoras a la aplicación implementada, además de convertirse en un inicio para la **Escuela Politécnica de Cáceres** en el uso de las tecnologías utilizadas para la creación del producto final, sobre todo el uso de contenedores y la orquestación de los mismos, ya que son la vía para la obtención de un sistema altamente automatizado que además abarate los gastos de mantenimiento y disminuya la necesidad de adquisición de nuevas infraestructuras.

En el ámbito personal, el desarrollo de este proyecto ha conllevado una gran adquisición de conocimientos para finalizar mis estudios de grado en la **Universidad de Extremadura**, ya que en el he tenido que utilizar muchos de los conceptos aprendidos durante el **Grado de Ingeniería Informática en Ingeniería de Computadores** y también me ha sido necesario aprender otra gran cantidad de ellos para poder alcanzar el objetivo final de este proyecto. He tenido que realizar un gran esfuerzo para aprender y utilizar tecnologías que eran desconocidas para mi, las cuales creo que serán de gran ayuda en un futuro para desempeñar un buen trabajo en el ámbito laboral. Para finalizar, he de destacar que ha sido muy satisfactorio y motivador poder conseguir los objetivos propuestos.

Capítulo 8

Futuros trabajos

Como posibles líneas de actuación para esta aplicación hay que mencionar:

- Relativas a la **API REST KubeKVM**:
  - Introducir un login y almacenar diferentes usuarios de la aplicación en la base de datos para proveer un servicio personalizado a cada uno, es decir que a cada usuario registrado se le muestren los datos relativos a los despliegues que ese usuario haya creado.
  - Introducir un registro que mantenga información de los accesos y acciones realizadas sobre el servicio.
- Relativas al script KVM **main.py**:
  - La creación de nuevas imágenes de máquinas virtuales básicas que cuenten con las tecnologías necesarias para dar soporte a más asignaturas impartidas por los docentes de la Escuela Politécnica de Cáceres.
  - La eliminación periódica de los despliegues que lleven un tiempo determinado sin utilizarse. Esta modificación se valdría de la modificación superior relativa al introducir un log a la API REST KubeKVM para consultar la información relativa al tiempo transcurrido desde la creación de un despliegue.

- Relativas al master del cluster de **Kubernetes**:
  - Definir un controlador de ingresos para realizar el emparejamiento de una url de http con el servicio definido para acceder a este mediante el nombre creado para la dirección en vez de utilizar la ip y el puerto ofertados por el servicio.
  - Definir un selector de nodos para que los pods sean creados en los nodos definidos en lugar de dejarlo a criterio del master del cluster de Kubernetes.
  - Definir un volumen persistente de datos para mantener almacenada la información introducida en los pods de Kubernetes tras eliminar el cluster.
  - Definir un autoescalador horizontal de pods para escalar automáticamente la cantidad de pods según la utilización observada de la CPU.
  - Introducir una prueba de deteccion para comprobar si el pod se esta ejecutando y una prueba de preparación para comprobar si el pod esta listo para atender solicitudes.

Anexos

Anexos A

Anexo I: Configuración inicial para el desarrollo de la API

Con el fin de poder realizar el desarrollo de la API es necesaria la instalación de un conjunto de tecnologías dentro del sistema operativo utilizado que es Ubuntu 16.04 LTS. Estas tecnologías permitirán el desarrollo tanto del front-end como del back-end de la API, al igual que la conexión con la base de datos utilizada.

Inicialmente necesitamos un **IDE**, para esto se instala un editor de código fuente de código abierto **Atom**, esto es opcional si ya se cuenta con editores que cumplan el propósito de trabajar con el lenguaje necesario para cada componente del sistema final.

Para esto se abre un terminal y se ejecuta el siguiente código:

```
$ sudo apt-get update
$ sudo apt-get install atom6
```

Tras esto, es necesario instalar el entorno en tiempo de ejecución multiplataforma **Node.js**, el cual sera utilizado de base para realizar el desarrollo de la API. Esto lo realizamos lanzando los siguientes comandos en el terminal:

```
$ sudo apt-get update
$ sudo apt-get install nodejs
```

Para comprobar la instalación y la **versión** de Node.js ejecutamos en un terminal:

```
$ node -v
```

```
v4.2.6
```

Figura A.1: Mensaje que informa de la versión de Node.js instalada.

También es necesario instalar el gestor de paquetes **npm**, el cual permitirá instalar fácilmente módulos y paquetes para usar con Node.js:

```
$ sudo apt-get install npm
```

Ahora es el momento de comenzar el proyecto de Node.js, para esto nos movemos a la carpeta creada para albergar dicho proyecto y en el terminal escribimos:

```
$ sudo npm init --yes
```

Este comando crea dentro de la carpeta un archivo llamado **package.json** que describe todo el proyecto.

```
{
  "name": "AppVisual",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node src/index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "nodemon": "^1.18.7"
  },
  "dependencies": {
    "connect-flash": "^0.1.1",
    "express": "^4.16.4",
    "express-handlebars": "^3.0.0",
    "method-override": "^3.0.0",
    "mongoose": "^5.3.15"
  }
}
```

Figura A.2: Contenido package.json de nuestra aplicación.

Procedemos a instalar los **módulos** que necesitamos para dotar a la API de las funcionalidades necesarias para su correcto funcionamiento. Para esto, escribimos en el terminal:

```
$ npm i express express-handlebars method-override mongoose
    connect-flash nodemon
```

Podemos ver todos estos módulos instalados en la sección **dependencies** que puede observarse en la imagen anterior del archivo Node.js.

Una vez realizadas todas las instalaciones anteriores, ya disponemos de todo lo necesario para realizar el desarrollo de la API con conexión a **MongoDB** a través de **Node.js**.

Anexos B

Anexo II: Configuración inicial para el desarrollo del script KVM

Para realizar el desarrollo del script KVM **main.py** desarrollado en lenguaje **Python** que funcionando como un demonio consumirá de la base de datos los datos de los diferentes despliegues creados a través de la API y procederá a la creación de las máquinas virtuales mediante el uso de la tecnología de virtualización **KVM**, ha sido la necesaria la instalación de varios tipos de tecnologías al igual que la creación de las imágenes a partir de las cuales han de crearse las máquinas virtuales necesarias en cada caso para satisfacer las necesidades del cada despliegue.

Inicialmente, es necesario instalar el lenguaje de programación **Python** mediante el uso del siguiente comando a través del terminal:

```
$ sudo apt-get install python2.7
```

Para comprobar la versión instalada lanzamos:

```
$ python2.7 --version
```

Vemos la versión instalada en el sistema de dos maneras diferentes:

- Escribiendo en el terminal:

```
$ python
```

De manera que entraremos en el interfaz de programación de Python a través de terminal.

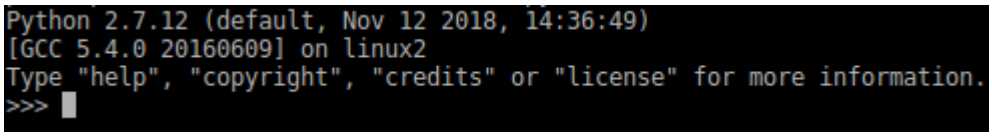


Figura B.1: Interfaz Python en el terminal.

- Escribiendo en el terminal:

```
$ python2.7 --version
```

Recibimos la versión de Python instalada .

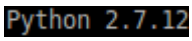


Figura B.2: Interfaz Python en el terminal.

Tras esto ya tenemos instalado el lenguaje que vamos a utilizar para el desarrollo del script que funcionará como un demonio consumiendo los datos almacenados por la API en la base de datos. Ahora es necesario instalar si es necesario KVM y crear las imágenes básicas de las máquinas virtuales a partir de las cuales serán creadas las diferentes máquinas virtuales requeridas por cada tipo de despliegue.

Para le ejecución de **KVM** se necesita un procesador con las extensiones de virtualización x86. Lo comprobamos de la siguiente manera.

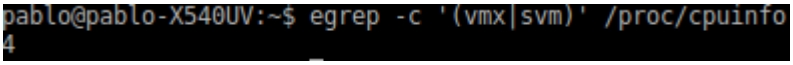


Figura B.3: Comprobación soporte de virtualización para KVM.

Al obtener una respuesta con un número mayor que 0, sabemos que el sistema tiene soporte de virtualización para KVM. Pero también hay que comprobar que esté activo en la BIOS.

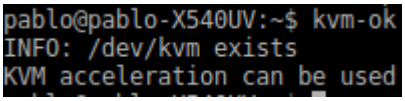


Figura B.4: Comprobación KVM activo en la BIOS.

Para virtualización con KVM se recomienda que tanto el procesador como el sistema Linux utilizado sea de 64 bits. A continuación verificamos la arquitectura.

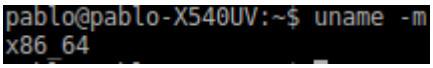


Figura B.5: Verificación arquitectura.

Tras realizar las comprobaciones anteriores, procedemos a instalar los paquetes necesarios para la configuración y manejo de esta tecnología con el fin de obtener el comportamiento requerido para nuestro producto final.

Primero instalamos los siguientes paquetes básicos:

- **qemu-kvm:** sistema KVM. Incluyendo el módulo del kernel.
- **libvirt-bin:** toolkit C para la interacción con el hipervisor.
- **bridge-utils:** herramientas para la configuración de bridges Ethernet.

Para ello ejecutamos el siguiente comando:

```
$ sudo apt-get install qemu-kvm libvirt-bin bridge-utils
```

Tras esto instalamos software adicional para facilitar la interacción y configuración de KVM:

- **virtinst:** proporciona herramientas en línea de comandos para la creación y clonado de VMs.
- **virt-manager:** interfaz gráfica para la gestión de VMs.

- **ubuntu-vm-builder:** scripts para la automatización de la creación de VMs basadas en Ubuntu.
- **virt-viewer:** permite la conexión a la consola de la máquina virtual a través del protocolo VNC.
- **virt-top:** permite obtener información estadística y de consumo de recursos para las máquinas virtuales de nuestra infraestructura Qemu/KVM.
- **libguestfs-tools:** es una biblioteca de C y un conjunto de herramientas para acceder y modificar imágenes de discos virtuales que se utilizan en la virtualización de plataformas.

Para ello ejecutamos el siguiente comando:

```
$ sudo apt-get install virtinst virt-manager ubuntu-vm-builder
    virt-viewer virt-top libguestfs-tools
```

Para terminar cargamos y activamos el módulo **vhost-net**.

```
$ sudo modprobe vhost_net
$ sudo lsmod | grep vhost
$ echo "vhost_net" | sudo tee -a /etc/modules
```

Comprobamos que todo funciona correctamente utilizando el comando **virsh**.

```
pablo@pablo-X540UV:~$ virsh -c qemu:///system list
 Id    Nombre                         Estado
----------------------------------------------------
```

Figura B.6: Comprobación del correcto funcionamiento de la instalación.

Tras esto, es el momento de la creación y configuración de las imágenes de las máquinas virtuales básicas a partir de las cuales se realizara el clonado de las máquinas virtuales necesarias para cada tipo de despliegue creado almacenado en la base de datos por la API.

Podemos realizar la creación de las máquinas virtuales de dos maneras:

- Mediante el uso del comando **virt-install**.

```
virt-install --connect qemu:///system
             --virt-type=kvm
             --name VirtualMachine01
             --ram 1024
             --vcpus=2
             --disk path=/var/lib/libvirt/images/
                 VirtualMachine01.img,size=8
             --cdrom /var/lib/libvirt/images/ubuntu-12.04-
                 server-amd64.iso
             --os-type linux
             --os-variant=ubuntuprecise
             --graphics vnc,keymap=es
             --noautoconsole
             --network network=default
             --description "Ubuntu 12.04 Server"
```

Figura B.7: Ejemplo instalación Ubuntu 12.04 Server desde una imagen ISO.

- Mediante el uso del comando **virt-manager**, tras el cual podremos realizar la creación y configuración de la máquina de manera gráfica.

Figura B.8: Interfaz virt-manager para la interacción con las máquinas virtuales.

En nuestro caso, se ha utilizado **virt-manager** para la creación de los dos tipos de máquinas virtuales requeridas para los despliegues. Estas máquinas cuentan con la siguiente configuración:

- **Windows XP:** esta máquina cuenta con 1 CPU virtual, 512 MiB de memoria RAM, 4 GiB de almacenamiento en disco y el sistema operativo Windows XP Service Pack 3 de 32 bits. Para cumplir los requerimientos de software de esta máquina se ha instalado el programa **logisim-win-2.7.1** y el navegador **Mozilla Firefox**.

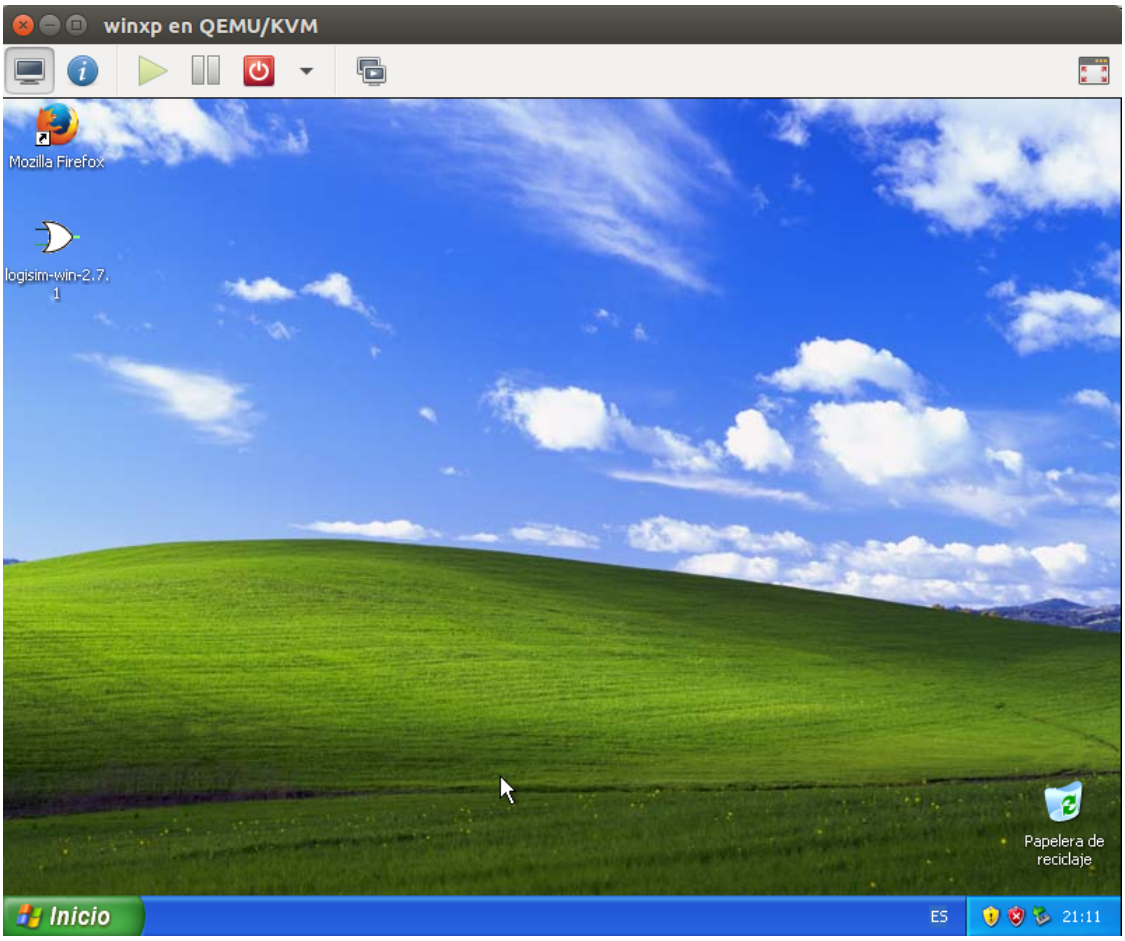


Figura B.9: Máquina básica Windows XP.

- **Ubuntu 16.04:** esta máquina cuenta con 1 CPU virtual, 1024 MiB de memoria RAM, 16 GiB de almacenamiento en disco y el sistema operativo Ubuntu 16.04 LTS 3 de 32 bits. Para cumplir los requerimientos de software de esta máquina se ha instalado el IDE de **Eclipse** con plugins para **C** y **C++**, y una distribución del lenguaje de programación **Python** y **R** llamada **Anaconda**, en particular se utilizara **Jupyter Notebook** que esta incluido dentro de este.

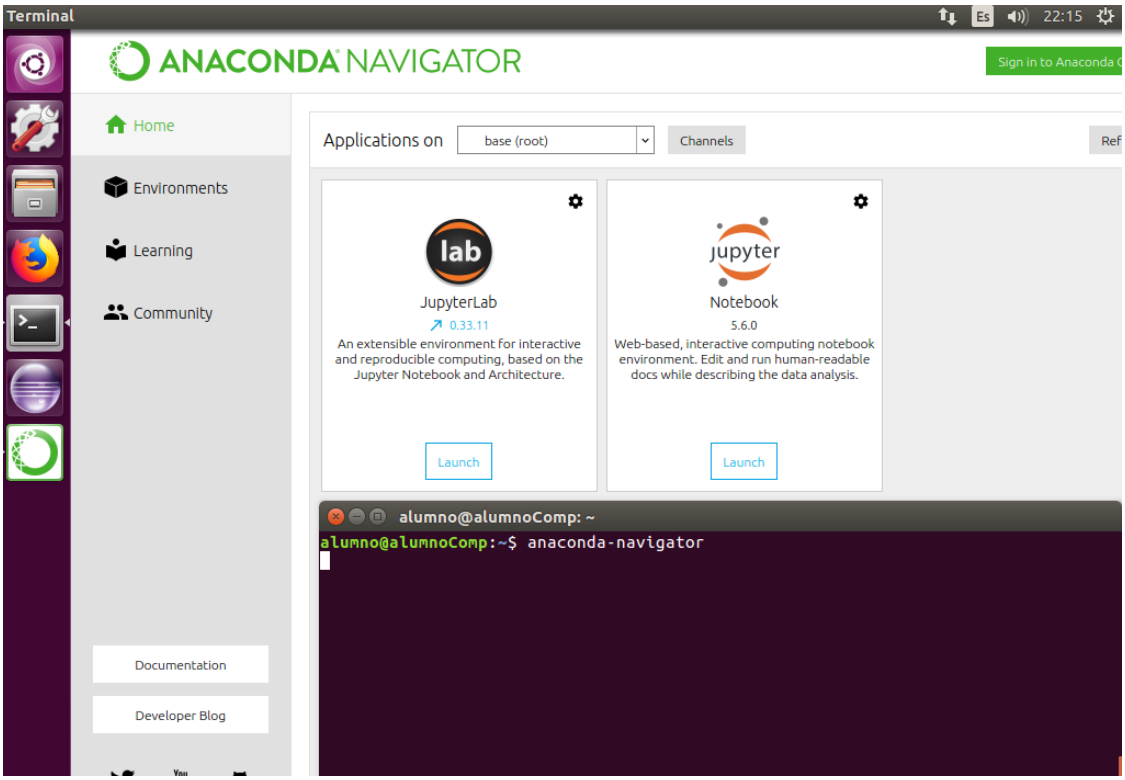


Figura B.10: Máquina básica Ubuntu 16.04.

Una vez creadas estas máquinas virtuales anteriores, ya disponemos de todo lo necesario para realizar el desarrollo del script KVM **main.py** que creará, modificará y manejará máquinas virtuales **KVM** e información almacenada en la base de datos **MongoDB**.

Anexos C

Anexo III: Configuración inicial para la creación del cluster de Kubernetes

Para poder obtener un cluster de **Kubernetes** en el que realizar la exposición de los servicios ofrecidos por la API, es necesaria la instalación y configuración de varias tecnologías sobre el cluster físico ubicado en la Universidad de Extremadura, específicamente **openshh-server**, **Docker**, **Kubernetes** y **Flannel**.

A la hora de realizar la instalación hay que realizar una diferenciación entre las máquinas instaladas en las hojas del cluster. Se disponen de 2 hojas en el cluster, cada una de ellas cuenta con el sistema operativo **Ubuntu 16.04 LTS**, estas máquinas serán utilizadas de la siguiente manera: una como el maestro y las otra será un nodo adicional.

El primer conjunto de instalaciones y configuraciones a realizar se harán sobre el **maestro**.

Antes de comenzar debemos apagar el swap, ya que Kubernetes lanza errores aleatorios durante su instalación:

```
# sudo apt-get update
# sudo swapoff -a
# sudo nano /etc/fstab
```

Dentro del fichero que se abre al lanzar el último comando, comentamos la siguiente línea.

```
#/dev/mapper/master--vg-swap_1 none            swap    sw              0       0
```

Figura C.1: Línea comentada en el fichero fstab.

Ahora debemos hacer referencia a los equipos en el cluster:

```
# sudo nano /etc/hosts
```

Configuramos el archivo con las direcciones IP como se ve en la imagen.

```
127.0.0.1       localhost
127.0.1.1       master
192.168.0.74    master
192.168.0.73    slave1
# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Figura C.2: Configuración archivo hosts con las IP de los equipos en el cluster.

Tras esto, debemos dotar al maestro de una IP estática en su segundo interfaz de red. Para realizarlo ejecutamos el comando:

```
# sudo nano /etc/network/interfaces
```

Dentro del archivo, introducimos las líneas que se ve en la imagen siguiente.

```
auto enp5s0
iface enp5s0 inet static
address 192.168.0.74
netmask 255.255.255.0
# gateway 192.168.0.1
```

Figura C.3: Configuración archivo interfaces con la IP estática.

Para poder realizar conexiones ssh con el equipo, procedemos a instalar **openshh-server** mediante la orden:

```
# sudo apt-get install openssh-server
```

El último paso antes de comenzar la instalación de Kubernetes es instalar **Docker** lanzando los siguientes comandos en el terminal:

```
        # sudo apt-get update
        # sudo apt-get install apt-transport-https ca-certificates curl
            software-properties-common
        # curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
            sudo apt-key add -
        # sudo apt-key fingerprint 0EBFCD88
        #  sudo add-apt-repository "deb [arch=amd64] https://download.
            docker.com/linux/ubuntu \
        $(lsb_release -cs) stable"
        # sudo apt-get update
        # sudo apt-get install docker-ce
```

Para comprobar la correcta instalación ejecutamos:

```
        # sudo docker ps
```

En la siguiente imagen vemos que la instalación se ha realizado de manera correcta.

Figura C.4: Respuesta al comando docker ps tras la correcta instalación de Docker.

Continuamos con las instalación de **Kubernetes** en el maestro. Para ello ejecutamos:

```
        # sudo apt-get update
        # sudo apt-get upgrade -y
        # sudo apt-get install apt-transport-https -y
        # sudo systemctl start docker
        # sudo systemctl enable docker
        # sudo curl -s https://packages.cloud.google.com/apt/doc/apt-
            key.gpg | sudo apt-key add -
        # sudo vi /etc/apt/sources.list.d/kubernetes.list
```

Dentro del fichero kubernetes.list introducimos la línea de la imagen que se observa a continuación.

Figura C.5: Línea para configuración introducida en kubernetes.list.

Proseguimos ejecutando comandos:

```
# sudo apt-get update
# sudo apt-get install -y kubelet kubeadm kubectl kubernetes-
    cni
```

Es el momento de cambiar el archivo de configuración de Kubernetes:

```
# nano /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
```

En el interior del fichero introducimos la siguiente linea después de la ultima variable de entorno como se ve en la imagen inferior:

```
# Environment="cgroup-driver=systemd/cgroup-driver=cgroupfs"
```

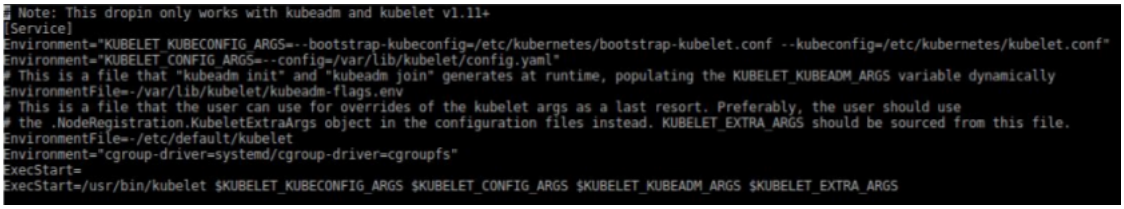


Figura C.6: Variable de entorno introducida en 10-kubeadm.conf.

Procedemos a iniciar el maestro ejecutando el comando:

```
# sudo kubeadm init --pod-network-cidr 10.244.0.0/16 --
    apiserver-advertise-address=192.168.0.74
```

Tras lanzar el comando anterior recibimos la siguiente imagen confirmando que el maestro de Kubernetes se ha iniciado correctamente junto al cluster.

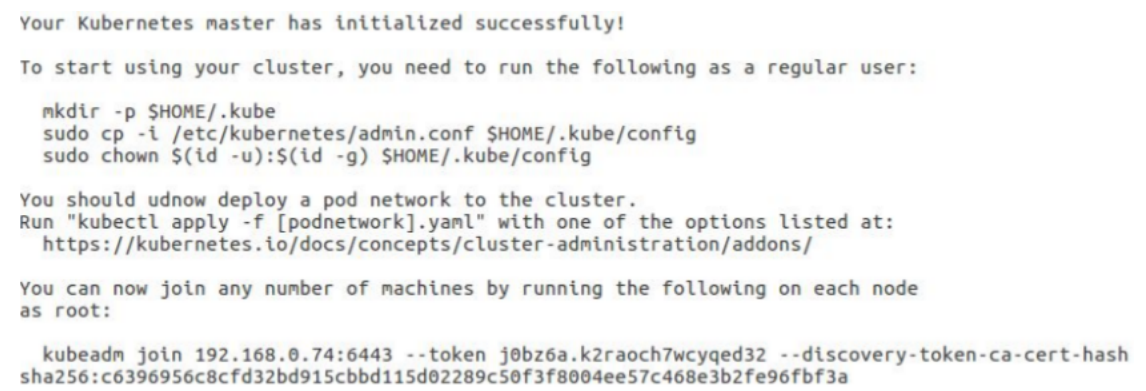


Figura C.7: Mensaje de inicio correcto del maestro.

En la imagen los comandos necesarios para comenzar a utilizar el cluster y también el comando que debemos lanzar en un nodo para que forme parte del mismo. Por tanto lanzamos los comandos para comenzar a utilizar el cluster:

```
# mkdir -p $HOME/.kube
# sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
# sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Para terminar, se requiere una red para permitir la comunicación entre los Pods que se desplegarán dentro de Kubernetes, para esto instalamos **Flannel**:

- Para que la Flannel funcione correctamente, se debe pasar –pod-network-cidr = 10.244.0.0/16 a kubeadm init. Establecemos /proc/sys/net/bridge/bridge-nf-call-iptables en 1 para pasar el tráfico IPv4 en puente a las cadenas de iptables ejecutando:

  ```
  # sudo sysctl net.bridge.bridge-nf-call-iptables=1
  ```

- Tras esto ejecutamos Flannel:

  ```
  # kubectl apply -f
  https://raw.githubusercontent.com/coreos/flannel/
      bc79dd1505b0c8681ece4de
  4c0d86c5cd2643275/Documentation/kube-flannel.yml
  ```

Una vez que tenemos el maestro del cluster de Kubernetes funcionando, es el momento de introducir un nodo en el mismo. El inicio de la configuración del nodo se realiza de la misma manera que con el maestro pero utilizando la IP designada para este nodo:

```
.......
Todo lo igual que con el maestro
.......
# sudo apt-get update
# sudo apt-get install -y kubelet kubeadm kubectl kubernetes-
    cni
```

Tras ejecutar los anteriores comandos es cuando cambian los comando a ejecutar para introducir al nodo en el cluster de Kubernetes. Primero introducimos el comando que recibimos en el nodo maestro al iniciarlo junto al cluster de Kubernetes:

```
## kubeadm join 192.168.0.74:6443 --token j0bz6a.
    k2raoch7wcyqed32
--discovery-token-ca-cert-hash
sha256:
    c6396956c8cfd32bd915cbbd115d02289c50f3f8004ee57c468e3b2fe96fbf3a
```

Puede darse el caso en el que el **token** o el certificado **hash** recibidos en el maestro hayan caducado, para obtenerlos de nuevo:

- Si necesitamos otro **token**, debemos conectarnos al maestro via ssh y escribir:

  ```
  # kubeadm token create
  ```

- Si necesitamos otro certificado **hash**, debemos conectarnos al maestro via ssh y escribir:

  ```
  # openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt |
      openssl rsa -pubin
  -outform der 2>/dev/null | \
  openssl dgst -sha256 -hex | sed 's/^.* //'
  ```

En la imagen siguiente vemos como realizar la comprobación de la inserción del nodo en el cluster mediante la ejecución del comando en el maestro vía ssh.

Figura C.8: Consulta nodos cluster Kubernetes.

Tras esto ya tenemos al nodo listo para desempeñar sus funciones en el cluster de Kubernetes.Para comprobar que Kubernetes ya esta funcionando correctamente junto a Flannel escribimos:

```
# sudo watch kubectl get pods --all-namespaces
```

Podemos observar que están funcionando todos los componentes de manera correcta y en qué maquina se están ejecutando.

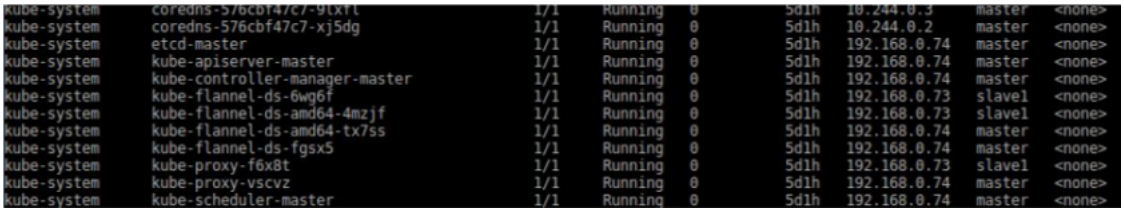


Figura C.9: Cluster de Kubernetes funcionando con todos sus componentes activos.

Una vez realizado todo lo anterior, ya disponemos de todo lo necesario para poder realizar el despliegue de nuestra **API** y **MongoDB** en el cluster de **Kubernetes**.

Anexos D

Agradecimientos

Me resulta muy difícil poder expresar expresar todos agradecimientos que siento que debería dar a todas las personas que me han apoyado y ayudado durante todos estos años.

A los primeros a las que debo dar mis agradecimientos son mis padres, las personas más importantes de mi vida, que me han dado todo lo que tenían para formarme como persona, que siempre han confiado en que yo podría conseguir esta meta. Nunca me ha faltado su apoyo y su cariño en este largo camino, Aunque hemos tenido algunos años difíciles para la familia y he estado varias veces apunto de tirar la toalla, junto a ellos he madurado y he aprendido que es lo realmente importante.

A mi hermano, por ser la mano que me ha acompañado desde la infancia, convirtiéndose en una figura a seguir y un ejemplo como persona. También debo mencionar a mi cuñada por ser la mujer con los mayores valores y principios que conozco y a mi sobrino por endulzarme cada segundo a su lado y hacerme ver todo el amor que profesan unos padres por sus hijos.

A los profesores de la **Escuela Politécnica de Cáceres** en la he adquirido tantos conocimientos y donde he ganado grandes amigos, en especial a Antonio Plaza Miguel por transmitirme su entusiasmo, su excelente trato y confianza, a Pablo García Rodríguez por su sencillez, cercanía y motivación, a Francisco Andrés Hernández que cuenta con toda mi gratitud y al que agradezco su ayuda constante e inestimable a pesar de mi pesadez en ciertos momentos y a mi tutor Antonio Manuel Silva Luengo uno de los principales artífices de mi continuidad en la universidad tras la enfermedad de mis padres, por su atención y por el gran corazón que alberga.

Para finalizar, a mis compañeros de tantos años de trabajo, esfuerzo y sobre todo grandes momentos a su lado, personas que tienen cuentan con mi amistad, toda mi confianza y que ya forman parte de mi vida para siempre, gracias a Jorge, Miguel Ángel, Carlos, Antonio Manuel, Alfonso, Jesus, Francisco, Juan Luis y Juan Francisco, gracias por todo.

Bibliografía

[1] Bootstrap, *Bootstrap Documentation*, https://getbootstrap.com/docs/4.2/getting-started/introduction/

[2] Connect Flash, *Connect Flash - npm*, https://www.npmjs.com/package/connect-flash

[3] CSS, *CSS Tutorial*, https://www.w3schools.com/css/

[4] Docker, *Docker Official Site*, https://www.docker.com/

[5] Express, *Express - Infraestructura de aplicaciones web Node.js*, https://expressjs.com/es/

[6] Express Handlebars, *Express Handlebars - npm*, https://www.npmjs.com/package/express-handlebars

[7] Flannel, *Flannel Documentation*, https://github.com/coreos/flannel#flannel

[8] HTML, *Hipertexto: El nuevo concepto de documento en la cultura de la imagen*, http://www.hipertexto.info/documentos/html.htm

[9] JavaScript, *JavaScript Official Site*, https://www.javascript.com/

[10] jQuery, *jQuery API Documentation*, https://api.jquery.com/

[11] JSON, *Introducción a JSON*, https://www.json.org/json-es.html

[12] Kubernetes, *Kubernetes Documentation*, https://kubernetes.io/docs/home/

[13] KVM, *KVM Documents*, https://www.linux-kvm.org/page/Documents

[14] Libvirt, *Libvirt Docs*, https://libvirt.org/docs.html

[15] Method Override, *Method Override - npm*, https://www.npmjs.com/package/method-override

[16] MongoDB, *Open Source Document Database | MongoDB*, https://www.mongodb.com/es

[17] Mongoose, *Mongoose Schemas*, https://mongoosejs.com/docs/guide.html

[18] Node.js, *Documentación Node.js*, https://nodejs.org/es/docs/

[19] Nodemon, *Nodemon Documentation*, https://github.com/remy/nodemon#nodemon

[20] Python, *Python Documentation*, https://www.python.org/doc/

[21] REST, *Introducción a Servicios REST*, https://www.arquitecturajava.com/servicios-rest/

[22] Virsh, *Virsh Command Reference*, https://libvirt.org/virshcmdref.html

[23] Virt Manager, *Virtual Machine Manager Official Site*, https://virt-manager.org/