

# **Ciclo Administración de Sistemas Informáticos en Red a Distancia**

**Curso 2021/22**

---

## **Proyecto**

**Kubernetes en la infraestructura local para  
el despliegue de una aplicación web**

---

Gabriel Cubillos Rodríguez

6 de junio de 2022

# Contenido

---

<b>Índice de figuras.....</b>	<b>2</b>
<b>Resumen.....</b>	<b>7</b>
<b>1. Introducción.....</b>	<b>8</b>
<b>2. Origen y contextualización.....</b>	<b>10</b>
2.1. El contexto: Kubernetes en el cloud.....	10
2.2. El problema: aplicación en el ámbito local.....	12
2.3. Una solución: Rancher.....	14
<b>3. Objetivos.....</b>	<b>16</b>
3.1. Objetivo principal.....	16
3.2. Etapas.....	16
<b>4. Planificación temporal.....</b>	<b>18</b>
<b>5. Arquitectura del clúster de Kubernetes.....</b>	<b>20</b>
5.1. Nodos.....	20
5.1.1. Definición y tipos.....	20
5.1.2. Componentes del nodo máster.....	21
5.1.3. Componentes del nodo worker.....	22
5.1.4. Propuesta.....	23
5.2. Pods.....	24
5.2.1. Definición.....	24
5.2.2. Propuesta.....	24
5.3. Servicios.....	25
5.4. Configuración de red.....	26
5.4.1. Propuesta: networking en K3s.....	27
<b>6. Estimación de recursos.....</b>	<b>29</b>
<b>7. Viabilidad y riesgos.....</b>	<b>30</b>
<b>8. Ejecución.....</b>	<b>31</b>
8.1. Creación de una aplicación web funcional.....	32
8.1.1. La vista.....	33
8.1.2. La base de datos.....	34
8.1.3. API.....	35
8.2. Crear imágenes Docker de la aplicación para subirlas al registro.....	36
8.3. Creación de un clúster de Kubernetes de seis nodos.....	39
8.3.1. Planteamiento.....	39
8.3.2. Balanceador de carga.....	40

8.3.3. Herramienta para instalación: K3sup.....	42
8.3.4. Preparación de las conexiones SSH.....	43
8.3.5. Creación del clúster: agregando el primer nodo.....	45
8.3.6. Gestión del clúster: Instalación de kubectl.....	47
8.3.7. Adición del resto de nodos.....	49
<b>8.4. Instalación de un balanceador interno: MetallLB.....</b>	<b>50</b>
<b>8.5. Despliegue de aplicaciones.....</b>	<b>52</b>
8.5.1. Volumen persistente de datos.....	53
8.5.1.1. Longhorn.....	55
8.5.2. Despliegue de la base de datos.....	58
8.5.2.1. Secret y ConfigMap.....	59
8.5.2.2. Creación de volúmenes de datos para MySQL.....	60
8.5.2.3. Aplicación del manifiesto.....	62
8.5.2.4. Un apunte sobre MySQL y Kubernetes.....	64
8.5.3. Despliegue de la API.....	66
8.5.4. Despliegue de la vista de la aplicación.....	68
8.5.5. Acceso a la web: Nginx Ingress Controller.....	70
<b>9. Evaluación.....</b>	<b>72</b>
<b>9.1. Operatividad de la aplicación.....</b>	<b>73</b>
<b>9.2. Comprobación de características de Kubernetes.....</b>	<b>76</b>
9.2.1. Escalabilidad.....	76
9.2.2. Alta disponibilidad.....	78
9.2.3. Balanceo de carga.....	81
<b>9.3. Monitorización.....</b>	<b>82</b>
9.3.1. Prometheus y Grafana.....	82
9.3.2. Panel de Kubernetes.....	84
9.3.3. Rancher.....	85
<b>ANEXOS.....</b>	<b>86</b>
<b>ANEXO I: Aplicación web.....</b>	<b>86</b>
Vista.....	86
API.....	97
Base de datos.....	102
<b>ANEXO II: Docker.....</b>	<b>104</b>
<b>ANEXO III: Utilidades.....</b>	<b>105</b>
<b>ANEXO IV: Manifiestos.....</b>	<b>106</b>
<b>Referencias.....</b>	<b>111</b>

## Índice de figuras

Figura 1: Un ejemplo de arquitectura de microservicios.....	8
Figura 2: Web de EKS.....	10
Figura 3: Ejemplo de facturación del servicio Google Kubernetes Engine.....	11
Figura 4: Documentación de K8s para Minikube.....	12
Figura 5: Panel de control de Rancher mostrando un clúster de tres nodos.....	14
Figura 6: Planificación temporal del proyecto.....	19
Figura 7: Detalle de la fase de planificación.....	19
Figura 8: Esquema de la relación entre los componentes de Kubernetes.....	22
Figura 9: Estructura del clúster de Kubernetes propuesto.....	23
Figura 10: Esquema de red del clúster que propone el proyecto.....	27
Figura 11: Esquema de configuración de red dentro del Pod.....	28
Figura 12: Tabla de requerimientos para la instalación de K3s.....	29
Figura 13: Detalle de dos filas en la aplicación.....	32
Figura 14: Vista de la aplicación Web.....	33
Figura 15: Esquema ER de la base de datos propuesta.....	34
Figura 16: Resultado de la petición GET de un módulo desde consola.....	35
Figura 17: Captura del repositorio de Docker Hub.....	36
Figura 18: Instalación de K3sup (elaboración propia).....	42
Figura 19: Archivo de configuración de alias SSH.....	43
Figura 20: Estado del servicio K3s en el nodo master1.....	46
Figura 21: Resaltado de línea modificada en el archivo <code>~/.kube/config</code> .....	47
Figura 22: Información de los nodos del clúster con kubectl.....	48
Figura 23: Lista de nodos del clúster ya completado.....	49
Figura 24: Un servicio tipo LoadBalancer con sus IP externas.....	50
Figura 25: Salida en consola de todos los objetos del namespace de MetallLB.....	50
Figura 26: ConfigMap para configurar MetallLB.....	51
Figura 27: Manifiesto para un volumen persistente de tipo nfs.....	53
Figura 28: Manifiesto para un PVC.....	54
Figura 29: El servicio longhorn-frontend se encarga de la interfaz gráfica de Longhorn.	56
Figura 30: El servicio responsable de la interfaz cambiado a LoadBalancer.....	56
Figura 31: El servicio longhorn-frontend con IP externa.....	57
Figura 32: Interfaz gráfica web de Longhorn.....	57
Figura 33: Secreto de la base de datos.....	59
Figura 34: ConfigMap de la base de datos.....	59
Figura 35: Menú de creación de un PV en Longhorn.....	60
Figura 36: Menú de creación de un PVC en Longhorn.....	60
Figura 37: Registro de un volumen vinculado en la interfaz de Longhorn.....	61

Figura 38: Archivo yaml generado por Longhorn.....	61
Figura 39: Manifiesto para el servicio MySQL.....	62
Figura 40: Detalle del volumen para MySQL en la interfaz de Longhorn.....	64
Figura 41: Manifiesto de despliegue de la API.....	66
Figura 42: Manifiesto para el servicio de la API.....	67
Figura 43: Listado de servicios del espacio de nombres por defecto.....	67
Figura 44: Llamada a la API en la consola.....	67
Figura 45: Manifiesto para el despliegue de la página web.....	68
Figura 46: Todos los Pod de la aplicación web.....	68
Figura 47: Manifiesto para el servicio de la vista Web.....	69
Figura 48: Servicios de NGINX Ingress Controller.....	70
Figura 49: Manifiesto Ingress.....	70
Figura 50: Información sobre el Ingress de la aplicación.....	71
Figura 51: Acceso a la aplicación Web desplegada en el clúster.....	71
Figura 52: Aplicación web recuperando un registro de la base de datos.....	73
Figura 53: Registro insertado a través de la aplicación web.....	73
Figura 54: Recuperación de un registro para su modificación.....	74
Figura 55: Tabla con el registro modificado.....	74
Figura 56: Selección de un registro para eliminar.....	75
Figura 57: Tabla con un registro ya eliminado.....	75
Figura 58: Rélicas de la API.....	76
Figura 59: Aumento de rélicas de la API.....	76
Figura 60: Salida de "kubectl describe deployment flaskapi-deployment".....	77
Figura 61: Información sobre la escalabilidad horizontal del despliegue.....	77
Figura 62: Log de eventos de vista-deployment.....	77
Figura 63: Número de nodos antes de la prueba de alta disponibilidad.....	78
Figura 64: Número de Pods antes de la prueba de alta disponibilidad.....	78
Figura 65: Estado del clúster al apagar el nodo worker2.....	78
Figura 66: Reprogramación de Pods por caída de un nodo.....	79
Figura 67: El clúster funcionando con un único nodo worker.....	79
Figura 68: Estado de los despliegues tras la caída de nodos.....	79
Figura 69: Estado de los despliegues ya recuperados.....	79
Figura 70: Reprogramación de Pods tras la caída del segundo nodo.....	80
Figura 71: Demostración del balanceo de carga de clúster.....	81
Figura 72: Servicios del espacio de nombres "monitoring".....	82
Figura 73: Mapeo de puertos para entrar en la UI de Grafana.....	82
Figura 74: Página de acceso de Grafana.....	83
Figura 75: Sección para agregar una fuente de datos en Grafana.....	83
Figura 76: Selección de paneles en Grafana.....	83

Figura 77: Kubernetes Dashboard funcionando en el clúster.....	84
Figura 78: Detalle de los Pods en Kubernetes Dashboard.....	84
Figura 79: Pantalla de monitorización de Rancher.....	85

## Índice de tablas

Tabla 1: Planificación temporal.....	18
Tabla 2: Configuración de Pods propuesta.....	24
Tabla 3: Relación de peticiones HTTP y consultas MySQL en la tabla "modulo".....	35

## Resumen

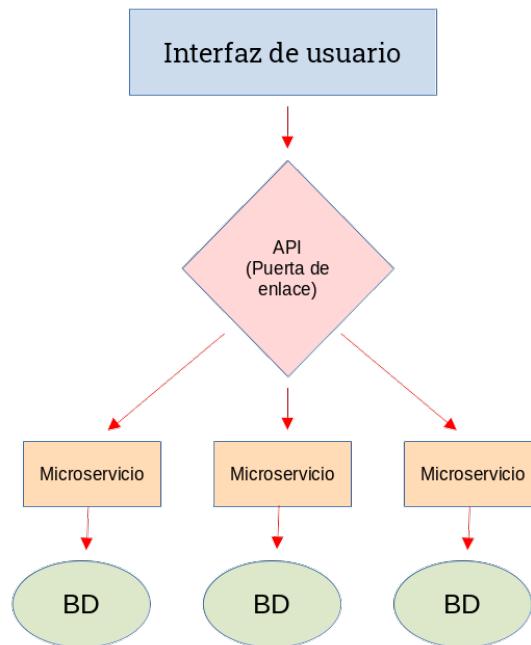
Este proyecto defiende la tesis de que la implementación de Kubernetes en infraestructura local es viable para entornos de producción en pequeñas y medianas empresas. Para ello, se creará en un entorno local un clúster conteniendo varios nodos, y se desplegará en ellos una aplicación web que constará de una vista, un controlador API y una base de datos. Finalmente, se evaluará el funcionamiento de la aplicación web, y se monitorizará la actividad del clúster en términos de rendimiento, balanceo de carga y disponibilidad.

# 1. Introducción

Kubernetes es un orquestador de contenedores de código abierto desarrollado por Google. Gestiona aplicaciones en cientos e incluso miles de contenedores en distintos entornos, ya sea en máquinas virtuales, físicas o entornos de cloud computing, como las plataformas de Amazon o Google.

El incremento de tecnologías para el manejo de contenedores, y más concretamente el auge de Kubernetes está relacionado con la transición desde el paradigma de desarrollo de arquitectura monolítica hacia la arquitectura de microservicios[1]:

- La arquitectura monolítica es aquella en la que el software se estructura de forma que todos los aspectos funcionales del mismo quedan sujetos en un mismo programa. La información necesaria para el trabajo de este tipo de sistemas queda alojada de forma estable en un único servidor.
- Los microservicios son un enfoque arquitectónico y organizativo para el desarrollo de software donde éste se compone de pequeños servicios independientes comunicados a través de API bien definidas. En este caso, el software no se presenta como una entidad individual, sino que cada una de sus funciones responde de forma individual y autónoma respecto de las otras.



*Figura 1: Un ejemplo de arquitectura de microservicios*

Los microservicios son, por tanto, aplicaciones pequeñas e independientes. Y es este rasgo el que convierte a los contenedores, por sus características de portabilidad, autosuficiencia y aislamiento, en la opción ideal para alojarlos.

## 2. Origen y contextualización

### 2.1. El contexto: Kubernetes en el cloud

De acuerdo con la información disponible en su página oficial [2], Kubernetes, también conocido como k8s, es "una plataforma de código abierto para automatizar la implementación, el escalado y la administración de aplicaciones en contenedores".

En términos más prácticos, podemos entender Kubernetes como un servicio en la nube ofrecido por las plataformas de *cloud computing* para desplegar y gestionar microservicios. Así, las grandes empresas de la tecnología de la información ofrecen servicios de Kubernetes administrados. Por ejemplo, en AWS, la nube de Amazon, podemos contratar Elastic Kubernetes Service (EKS) [3], y en la de Google, Google Kubernetes Engine (GKE) [4].



Figura 2: Web de EKS

Procedencia: <https://aws.amazon.com/es/eks/>

La principal ventaja de este tipo de producto es doble:

1. La administración y el mantenimiento de los nodos controladores del clúster de Kubernetes corre a cargo de la plataforma que presta el servicio, de forma que el cliente puede dedicarse exclusivamente al despliegue de aplicaciones en los *workers* o nodos de trabajo.
2. Las ventajas propias de un servicio en la nube, y principalmente: la asignación automática de una IP pública a cada servicio que se deseé exponer, el despliegue automatizado de平衡adores de carga sin apenas intervención del usuario y la integración de la gestión de CI/CD (integración continua y entrega continua) [5].

El coste de este producto varía conforme a la dimensión de la infraestructura contratada y de su disponibilidad a lo largo del tiempo. En este aspecto, la propia naturaleza del orquestador de contenedores implica la disminución o incremento de la disponibilidad de servicios según su demanda, lo que contribuye a optimizar en cierta medida los costes. Otra desventaja que no debemos eludir está relacionada con la pérdida de *autonomía digital* a consecuencia de delegar la gestión de nuestros servicios en otra empresa; quedamos expuestos a cambios en los términos de contratación que pueden afectar a los costes o a la productividad de la empresa.

Cuentas de facturación de la organización	Horas de clústeres de Autopilot al mes	Horas de clústeres regionales al mes	Horas de clústeres de zona al mes	Crédito del nivel gratuito utilizado	Tarifa mensual total de gestión de clústeres de GKE (0,1 USD por hora y por clúster)
cuenta_1	744	0	0	74,40 USD	0 USD
cuenta_2	0	1000	500	50 USD	100 USD
cuenta_3	1000	1000	1000	74,40 USD	225,60 USD

Figura 3: Ejemplo de facturación del servicio Google Kubernetes Engine  
Procedencia: Captura de <https://cloud.google.com/kubernetes-engine/pricing>

Como se puede deducir de estas características, con sus pros y contras, este producto es especialmente interesante para empresas de desarrollo de software y aquellas dedicadas a la implementación de microservicios.

Ahora bien, **¿qué ocurre con aquellas empresas o instituciones cuya estrategia de desempeño no incluye la contratación de servicios de cloud? ¿Deben renunciar a esta tecnología?**

Este proyecto pretende dar respuesta a esta pregunta.

## 2.2. El problema: aplicación en el ámbito local

Las principales ventajas de implementar Kubernetes en la infraestructura local son las siguientes:

- Total control de la administración del clúster.
- Posibilidad de aislar los sistemas y servicios para su uso exclusivo en un entorno local.
- Solución más económica a corto y medio plazo que la contratación de un servicio en la nube.

Respecto de los inconvenientes, podríamos destacar:

- Es necesario hacerse cargo de la administración completa del clúster, es decir, de los nodos maestros, del mantenimiento de la base de datos del estado del clúster, etc...
- Mayor complejidad en el uso de algunas características, como la creación del balancador de cargas, o el procedimiento para exponer al exterior alguno de los servicios.
- Coste de adquisición de equipos y dispositivos, y mantenimiento de la infraestructura.

La documentación oficial ilustra el procedimiento de creación de un clúster de Kubernetes en equipos locales usando Minikube[6]. Este programa permite la creación de una máquina virtual en un disco local donde el usuario podrá crear el clúster de Kubernetes.

The screenshot shows a section of the Kubernetes Documentation titled "Create a Cluster". The URL in the address bar is <https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/>. The page content includes a heading "Create a Cluster", a subtext "Learn about Kubernetes cluster and create a simple cluster using Minikube.", and two links: "Using Minikube to Create a Cluster" and "Interactive Tutorial - Creating a Cluster".

Figura 4: Documentación de K8s para Minikube  
Procedencia: <https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/>

Minikube es un sistema especialmente útil para el entorno de pruebas, el aprendizaje y la práctica de Kubernetes, en concreto de la utilidad de línea de comandos `kubectl`. Sin embargo, visto que se trata del despliegue de nodos en un sólo disco o *host*, no es un sistema apropiado para entornos de producción.

Resumiendo, encontramos aquí un problema: la documentación de Kubernetes no ofrece una solución sencilla o mínimamente práctica para el despliegue de un clúster en infraestructura local para entornos de producción.

## 2.3. Una solución: Rancher

Entre las soluciones que se encuentran en el ecosistema de software para la creación de un clúster de Kubernetes en local destaca Rancher[7]. Este software *open source* permite la creación de clúster remotos, locales o híbridos.

Como se verá en el desarrollo del proyecto, la creación de los nodos es bastante sencilla. Además, dispone de un panel de administración y monitorización que facilita la gestión de los diferentes nodos.

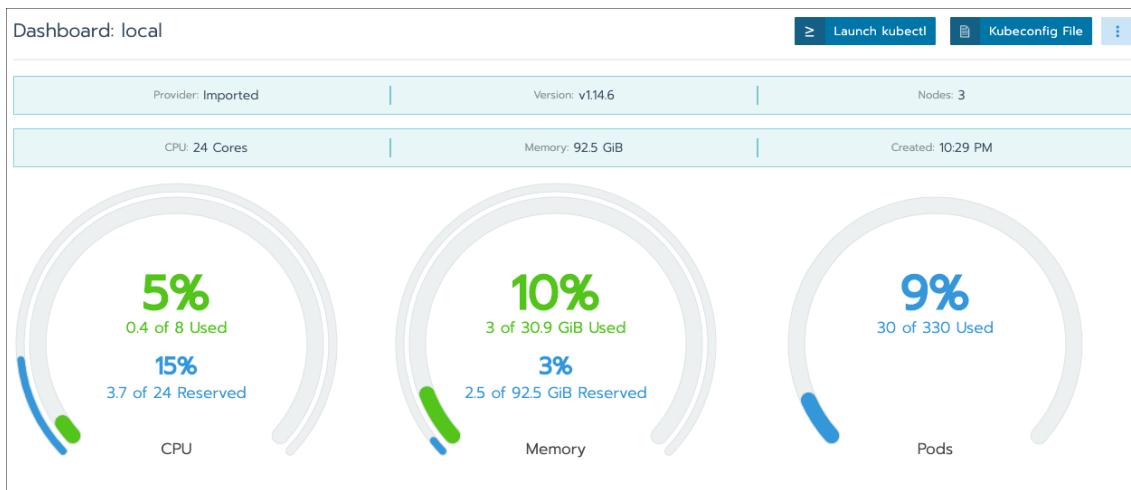


Figura 5: Panel de control de Rancher mostrando un clúster de tres nodos  
Procedencia: <https://github.com/rancher/rancher/issues/22746>

Hay que recordar que en todo momento estamos tratando con un caso de uso muy concreto: una empresa pequeña o mediana que implementa los servicios de su sistema informático de forma local, de forma que Rancher se configura como una solución idónea para evitar la complejidad de otros sistemas de creación de clúster de Kubernetes, a la vez que se elude la dependencia de plataformas de alojamiento externas.

Para la creación de clústeres, Rancher pone a disposición del usuario distintas distribuciones: RKE, RKE2 y K3s.

**K3s será la que emplearemos para conseguir los objetivos propuestos.**

K3s es una distribución de Kubernetes creada por Rancher que permite la creación de clústeres de alta disponibilidad. Se caracteriza por su ligereza: conforme a la documentación oficial[8], los requisitos mínimos del equipo para hacerla funcionar son 512 Kb de RAM con un núcleo de procesador virtual (VCPU).

Las formas de instalación varían según se desee instalar Rancher usando Docker para la creación del servidor principal, o crear el clúster manualmente agregando un servicio k3s

en cada nodo y, de forma opcional, posteriormente instalar la interfaz gráfica de Rancher para su gestión. Emplearemos la segunda opción, aunque más adelante profundizaremos en este punto.

## 3. Objetivos

### 3.1. Objetivo principal

Como ya se ha dicho, el objetivo principal del proyecto es el despliegue de una aplicación web en un entorno local de producción empleando una arquitectura de microservicios con el orquestador de contenedores Kubernetes.

### 3.2. Etapas

La realización del proyecto se estructura en distintas etapas:

#### 1. Documentación y pruebas preliminares:

- 1.1. Documentación sobre las distintas distribuciones de Kubernetes para la creación de un clúster en local.
- 1.2. Realización de pruebas preliminares relativas a la estabilidad de las distintas alternativas para la creación del clúster para determinar la más adecuada al objeto del proyecto.

#### 2. Creación de la aplicación web:

- 2.1. Diseño y creación de la vista de usuario empleando HTML5 y Javascript.
- 2.2. Diseño y creación de la base de datos relacional (MariaDB).
- 2.3. Diseño y creación de una API que atienda las peticiones del usuario a la base de datos.

#### 3. Creación del clúster empleando la distribución de Kubernetes k3s en distintas máquinas locales, conforme a la siguiente arquitectura:

- Balanceador de carga de acceso a los nodos maestros.
- Tres nodos maestros.
- Tres nodos de trabajo.

#### 4. Despliegue de la aplicación web en el clúster de Kubernetes:

- 4.1. Creación de las imágenes que contengan las aplicaciones, usando Docker para ello.
- 4.2. Configuración de los volúmenes que se emplearán para la persistencia de los datos.

4.3. Preparación y ejecución de los manifiestos para crear los distintos Pods que contendrán las distintas aplicaciones y servicios y configurar la red para su acceso.

**5. Evaluación del desempeño del clúster a través de programas de monitorización para Kubernetes, especialmente:**

5.1. Instalación del componente de monitorización.

5.2. Comprobación del correcto balanceo de carga.

5.3. Comprobación del correcto funcionamiento de las características de alta disponibilidad.

## 4. Planificación temporal

La planificación temporal se estructura conforme a la anterior relación de objetivos.

Como se puede ver, la fase más extensa es la dedicada a la investigación y realización de pruebas con vistas a determinar qué distribución de Kubernetes de las múltiples que se encuentran disponibles se adapta a los criterios de fiabilidad y estabilidad que exige un entorno de producción, a la vez que permite la configuración de componentes necesarios para infraestructura local.

Proceso	Inicio	Duración	Fin
[Documentación]	18/04/22	39	27/05/22
Investigación	18/04/22	39	27/05/22
Pruebas preliminares	01/05/22	9	10/05/22
[Aplicación Web]	09/05/22	3	12/05/22
Vista	09/05/22	1	10/05/22
API	10/05/22	1	11/05/22
Base de datos	11/05/22	1	12/05/22
[Preparación del clúster]	12/05/22	8	20/05/22
Creación del clúster de Kubernetes	12/05/22	1	13/05/22
Creación de imágenes Docker y registro	13/05/22	7	20/05/22
[Despliegue]	20/05/22	8	28/05/22
Configuración de volúmenes de datos	20/05/22	2	23/05/22
Preparación de los manifiestos	23/05/22	5	28/05/22
[Comprobaciones]	28/05/22	2	30/05/22
Monitorización	28/05/22	1	29/05/22
Balanceo de carga y alta disponibilidad	29/05/22	1	30/05/22

Tabla 1: Planificación temporal

La investigación sobre los distintos componentes y procedimientos se extiende a lo largo del proyecto solapándose con el resto de fases según se alcanzan los distintos objetivos.

Podría llamar la atención que para la creación del clúster, un objetivo fundamental en este proyecto, se planifique solamente un día. Sin embargo, es la fase de previa de documentación y pruebas la que permite que el procedimiento de creación se realice de forma sencilla en poco tiempo.

A continuación se expresa la planificación en un diagrama de Gantt:



Figura 6: Planificación temporal del proyecto

Y con más detalle, sólo la fase de ejecución:

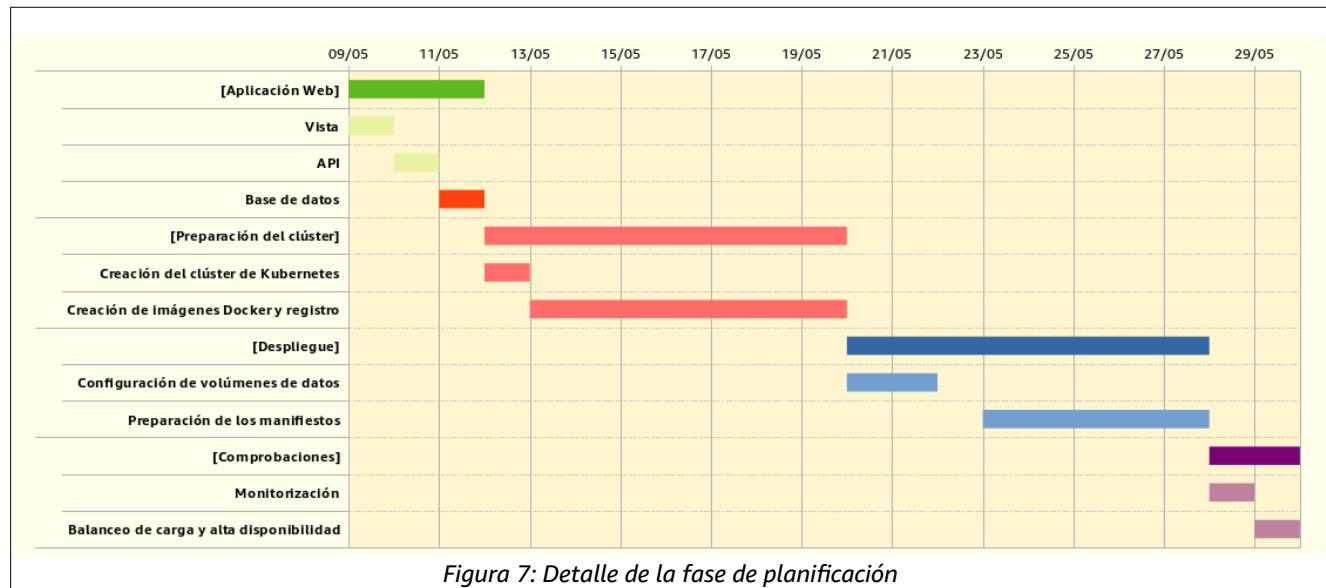


Figura 7: Detalle de la fase de planificación

## 5. Arquitectura del clúster de Kubernetes

### 5.1. Nodos.

#### 5.1.1. Definición y tipos

El nodo es una máquina de trabajo, que puede ser física o virtual. A diferencia de los Pods y servicios, los nodos no son creados por Kubernetes, sino por el administrador. Cuando se agrega un nodo al clúster, Kubernetes crea un objeto que representa al nodo, llamado *node*, y comprueba si es válido o no.

Conforme a las indicaciones de buenas prácticas, en un clúster de Kubernetes en infraestructura local se pueden distinguir dos tipos de nodos:

- **Nodos máster:** albergan los componentes que trabajan en el **plano de control**, o dicho de otro modo, en la gestión del clúster. Estos nodos sólo tienen sentido en un entorno local o, si es en remoto, servidores dedicados o *baremetal*. En los clúster de Kubernetes en la nube, los componentes del plano de control son gestionados por el proveedor de servicios correspondiente.
- **Nodos worker:** Alojan los Pods desplegados por el administrador, e incluyen los componentes relacionados con su funcionamiento y comunicación.

### 5.1.2. Componentes del nodo máster

Los componentes del plano de control, alojados en los nodo máster, son los siguientes:

- **kube-apiserver:** es el componente que expone la API de Kubernetes: recibe las peticiones y actualiza en consecuencia la base de datos etcd.
- **etcd:** se trata de una base de datos persistente, distribuida y formada por pares de clave-valor que guarda la información sobre el clúster y su estado.
- **kube-scheduler:** este componente decide en qué nodo se ejecuta cada réplica de un Pod.
- **kube-controller-manager:** este componente ejecuta los controladores de Kubernetes. Aunque cada controlador es independiente, se compilan en un único binario y se ejecutan en un sólo proceso. Estos controladores son:
  - **Controlador de nodos (*Node Controller*):** el responsable de detectar y responder cuando un nodo deja de responder.
  - **Controlador de replicación (*Job Controller*):** ante una nueva tarea se asegura de que en algún lugar del clúster los kubelets de un conjunto de nodos estén ejecutando la cantidad correcta de Pods para realizar el trabajo. No ejecuta ningún Pod por si mismo, sino que se comunica con la API para crear o eliminar Pods.
  - **Controlador de endpoints (*Endpoints Controller*):** construye el objeto de Kubernetes *Endpoints*, es decir, relaciona el acceso a un servicio con número de Pods.
  - **Controladores de tokens y cuentas de servicio:** gestionan las cuentas y tokens de acceso a la API por defecto para los nuevos espacios de nombres del clúster.

Aunque no es relevante para la propuesta de este proyecto, cabe mencionar como último componente el **cloud-controller-manager**, que ejecuta los controladores que interactúan con proveedores en la nube.

### 5.1.3. Componentes del nodo worker

Los nodos workers incluyen los siguientes componentes:

- **kubelet:** Es el encargado de ejecutar los contenedores dentro de cada Pod, asignándoles recursos del nodo conforme a la configuración establecida. Por tanto, es el componente que relaciona el entorno de ejecución de los contenedores con el nodo.
- **kube-proxy:** Es un proxy de red y balanceador de carga que dispone de la capacidad de reenviar el tráfico a otros nodos según los parámetros de red, establecidos por el administrador, que pueden permitir el acceso a los nodos desde el interior o el exterior del clúster. Por defecto, el algoritmo que emplea para el balanceo de carga es round-robin. De todo esto se desprende que el criterio que determina las conexiones entre distintos Pods no es la coincidencia dentro del mismo nodo. Por ejemplo, una petición desde la web originada en el nodo 3, en vez de dirigirse al Pod que contiene la API de ese mismo nodo puede ser atendida por la réplica API de cualquier otro nodo.
- **Entorno de ejecución de contenedores (*Container Runtime*):** Es el software responsable de ejecutar los contenedores. Kubernetes soporta Docker y containerd entre otros. En esta propuesta, el entorno de ejecución será containerd.

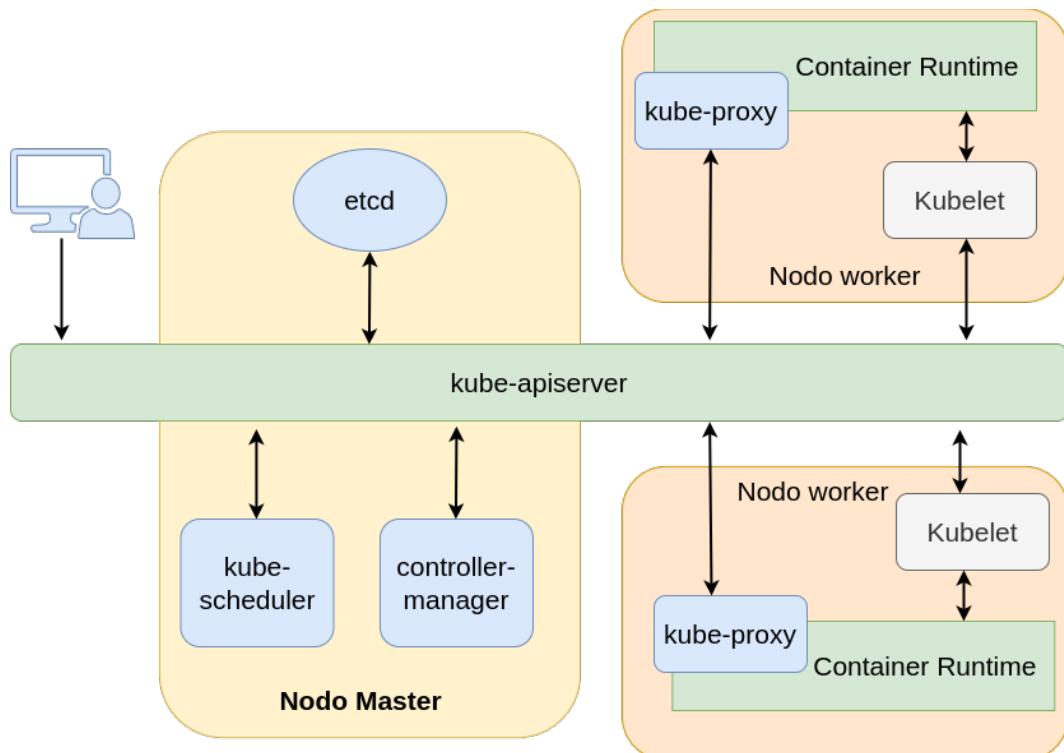
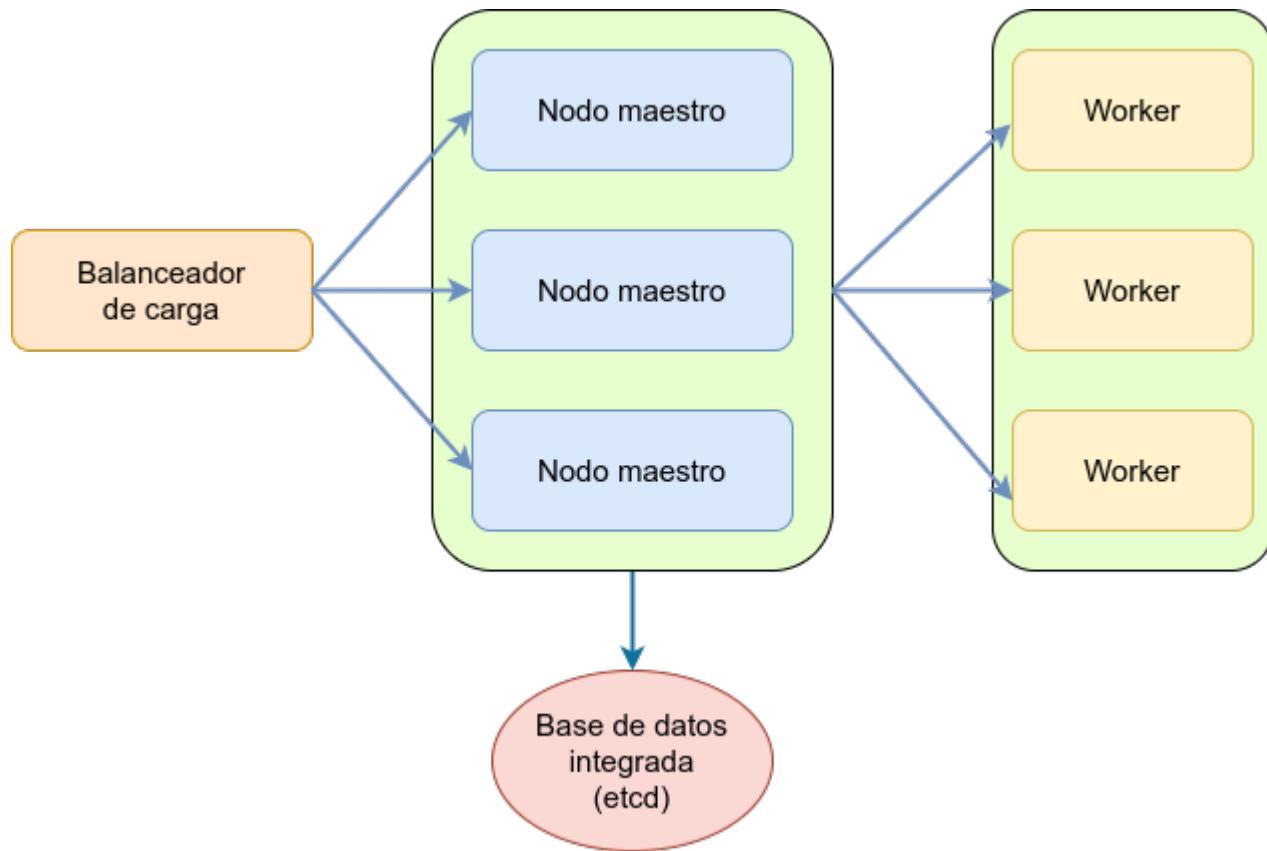


Figura 8: Esquema de la relación entre los componentes de Kubernetes  
(Elaboración propia)

### 5.1.4. Propuesta

El sistema propuesto consta de los siguientes elementos:

- Tres nodos máster o servidores.
- Tres nodos workers.
- Un balanceador de carga externo al clúster para el acceso a los nodos servidores.



*Figura 9: Estructura del clúster de Kubernetes propuesto  
(elaboración propia)*

Esta arquitectura responde a las recomendaciones de buenas prácticas descritas en la documentación para un clúster de alta disponibilidad, especialmente en lo que se refiere al número de nodos maestros, que se recomienda impar y mayor que uno para que exista la posibilidad de *quorum* en la gestión de los nodos trabajadores.

La base de datos que contiene la información sobre el estado de cada nodo del clúster en momento dado, estará integrada en éste y será de tipo etcd.

## 5.2. Pods

### 5.2.1. Definición

Un Pod es un grupo de uno o varios contenedores con la red y almacenamiento compartidos. Conforme a la documentación de Kubernetes, los Pods son las unidades de computación desplegables más pequeñas que se pueden desplegar y gestionar. Se trata de una capa de abstracción sobre uno o varios contenedores, de forma que el administrador del clúster opera sobre los Pod y no sobre contenedores.

Una de las características más importantes de los Pod es que son entidades efímeras. Su ciclo de vida está limitado por las políticas de reinicio o supresión como, por ejemplo, la superación de un concreto límite en el consumo de recursos del nodo. En estos casos, el Pod se suprime o “muere”, y el orquestador lo recrea en el mismo u otro nodo, cuidando de que existan el número de réplicas indicados en el manifiesto de despliegue.

### 5.2.2. Propuesta

Para el clúster propuesto en este proyecto se desplegarán nueve Pods. Cada Pod incluirá únicamente un contenedor:

- Servidor web: Es el único servicio expuesto al exterior. Se crearán tres réplicas.
- Servidor de base de datos: Servidor MySQL. Solamente establece conexión con la API. Contará con dos réplicas.
- Interfaz entre la base de datos y la web (API): Recibe las peticiones enviadas desde la Web y las remite la base de datos y devuelve el resultado. Se crearán cuatro réplicas.

Por tanto, tendremos que desplegar tres contenedores con sus tres servicios:

Contenedor	Expuesto al exterior	Réplicas (número de Pods)
Página web	Sí	3
Base de datos	No	1
API	No	2

Tabla 2: Configuración de Pods propuesta

Como se verá más adelante en la descripción de los manifiestos, cada uno de los Pods se despliegan indicando un número de réplicas. Los nodos maestros distribuyen los Pods entre los distintos nodos workers.

## 5.3. Servicios

El objeto de Kubernetes llamado Service, o servicio, es una abstracción que define un conjunto lógico de Pods y la política a través de la que se accede a ellos. Dicho de otra manera, si el Pod es una capa de abstracción sobre un grupo de contenedores, el Service es una capa de abstracción añadida sobre un grupo de Pods.

Como se dijo, los Pods no son recursos permanentes: se crean y destruyen de forma dinámica. En un clúster, cada réplica de un Pod cualquiera perteneciente a una aplicación dispone de su propia IP. El sentido que tiene el servicio es organizar el acceso a esos Pods de la aplicación. Si un agente interno o externo quiere acceder a la aplicación, tiene que existir un objeto que describa la forma en que se accede determinando, por ejemplo, los puertos o el balanceo de carga. Este objeto es el Service.

Existen servicios internos o externos, según el grado de exposición con que deseemos configurar el a la aplicación. Esta característica se define en el atributo "type" del manifiesto que crea el servicio. Para este atributo existen tres valores:

1. **ClusterIP**: Es un servicio interno. La IP no se expone fuera del clúster y sólo pueden acceder a estos Pods otros que hayan sido determinados para hacerlo. Este es el tipo de servicio que se asignará a la aplicación web y al gestor de base de datos.
2. **LoadBalancer**: es un servicio externo. El servicio dispone de una IP interna y una externa desde la que se puede acceder desde el exterior del clúster. La API desplegará este tipo de servicio.
3. **NodePort**: Este tipo de servicio se suele emplear para realizar pruebas de depuración. Abre un puerto en uno o varios nodos para acceder directamente a los servicios.

En la propuesta de este proyecto existirán tres servicios, uno para cada aplicación: Web, API y servidor MySQL. Como verá más adelante en la elaboración de los manifiestos, el vínculo entre el despliegue de Pods para una aplicación y el de su servicio es tan evidente y directo que para nuestro proyecto lo más práctico será configurarlo en el mismo archivo.

## 5.4. Configuración de red

La configuración de red en Kubernetes es un aspecto fundamental, y muchas veces el de mayor dificultad.

Kubernetes implementa el modelo "IP por Pod": las IP se dan en el ámbito y al alcance del Pod. Esto significa que los contenedores dentro del Pod comparten los espacios de nombres de red, la IP y la dirección MAC. Los contenedores pueden acceder a los puertos del resto de contenedores del Pod en el *localhost*.

¿Cómo reciben la IP los Pods y qué protocolo se usa para su comunicación? El modelo de configuración de red depende del CNI (Container Network Interface)[\[9\]](#) del clúster. Existen varios plugins de CNI. En nuestro caso, K3s integra por defecto Flannel[\[10\]](#), aunque existen otras alternativas muy extendidas, como Calico, que usa el protocolo BGP para la comunicación entre Pods.

Sea cual fuere el CNI, Kubernetes impone una serie de requisitos fundamentales en cualquier implementación de red:

- Los Pods en un nodo pueden comunicarse con todos los Pods en todos los nodos sin NAT.
- Los agentes de la API de Kubernetes en un nodo (como kubelet, por ejemplo) pueden comunicarse con todos los pods en ese nodo.
- Y para las plataformas que admiten Pods que se ejecutan en la red del anfitrión (por ejemplo, Linux), los Pods en la red host de un nodo pueden comunicarse con todos los Pods en todos los nodos sin NAT.

Por último, para acceder desde el exterior a un Pod a través de una URL, Kubernetes utiliza un componente llamado **ingress-controller**, que entrega los datos de la relación entre la dirección URL y el Service que gestiona el Pod al componente kube-proxy.

Resumiendo, podemos clasificar las comunicaciones dentro de la red de un clúster de la siguiente manera:

1. Comunicaciones de contenedor a contenedor en un Pod: resuelto por el Pod y las comunicaciones *localhost*.
2. Comunicaciones de Pod a Pod: determinado por el CNI que se implemente en el clúster.
3. Comunicaciones entre un Pod y un servicio: resuelto por el objeto Service.
4. Comunicaciones del exterior con un servicio: resuelto por el componente ingress-controller.

### 5.4.1. Propuesta: networking en K3s

Por defecto, K3s cuenta con los siguientes elementos para la configuración de red:

- CoreDNS: Un proveedor de DNS del clúster que se despliega en cada nodo worker.
- Traefik: se configura como proxy inverso, en este caso empleado para el enruteamiento y el balanceo de carga desde el exterior del clúster desde nombres de dominio hacia los servicios del clúster. Usaremos, en cambio, Nginx Ingress Controller
- Balanceador de carga Klipper para los servicios. Nosotros usaremos MetallLB
- CNI Flannel: utiliza túneles VXLAN para que los Pods de los diferentes nodos puedan conectarse sin NAT.

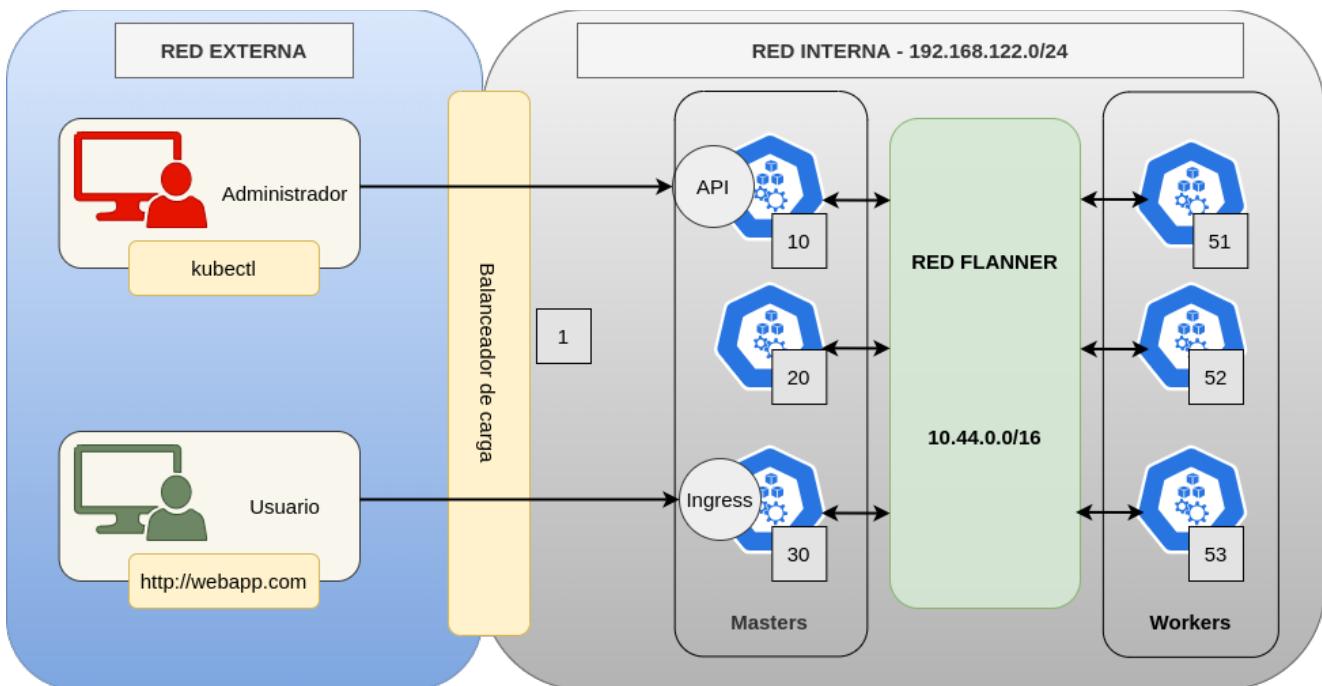


Figura 10: Esquema de red del clúster que propone el proyecto  
(Elaboración propia)

De la figura anterior se puede deducir lo siguiente:

- El administrador del clúster accede a él con kubectl, una herramienta de línea de comandos que sirve para interactuar con la API, y que debe estar configurada previamente para apuntar a la IP de la red externa del balanceador de carga.
- El usuario ingresa en su navegador la URL que resolverá la IP del balanceador de carga en la red externa, que a su vez lo encaminará hacia los nodos máster para su procesado a través del ingress-controller.
- La red interna 192.168.122.0/24 es la red que comparten los nodos.

- La red flanner 10.44.0.0/16 es una red virtual creada por el CNI para que los Pods se comuniquen entre sí.

Dentro de cada nodo, cada Pod dispondrá una IP de la red Flanner:

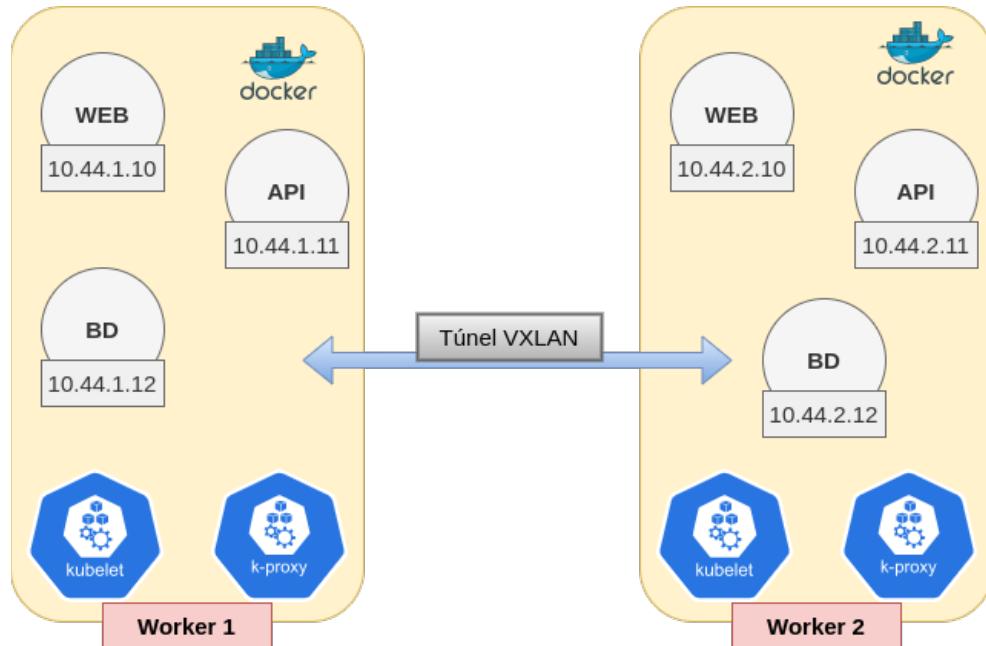


Figura 11: Esquema de configuración de red dentro del Pod  
(Elaboración propia)

## 6. Estimación de recursos

Los recursos necesarios para mantener en funcionamiento un clúster de Kubernetes con una aplicación web en entorno en producción vienen determinados fundamentalmente por dos tipos de factores:

- Factores internos: determinados por la arquitectura del clúster que va a dar soporte al sistema, es decir, el número de nodos, la estimación de servicios que pueden correr simultáneamente, los recursos asignados a cada uno de ellos, etc.
- Factores externos: un gran número de conexiones de usuarios para consultar la aplicación web puede determinar puntualmente el incremento de requerimientos para que el clúster se mantenga operativo.

Los factores internos se concretan en la arquitectura propuesta para la creación del clúster: tres nodos servidores o maestros (*master*) y otros tres nodos *workers*.

Los recursos disponibles para cada servicio se establecen en los manifiestos que los despliegan. Es habitual "sobrevender" los recursos del clúster, es decir, asignar recursos a los servicios de forma que sumados todos ellos superen los recursos reales de las máquinas que soportan los nodos. Esto se hace así porque es improbable que todos los servicios soliciten el máximo de recursos en un mismo instante.

En relación con el clúster que propone este proyecto, aunque los requerimientos mínimos que exige K3s podrían ser suficientes para una carga baja (por ejemplo, un servidor web para una oficina), lo aconsejable en previsión de picos de carga de trabajo, sería al menos 2 GB de RAM para cada nodo, es decir, un total de 12 GB si los nodos se instalan en la misma máquina y 2 VCPU.

DEPLOYMENT SIZE	NODES	VCPUS	RAM
Small	Up to 10	2	4 GB
Medium	Up to 100	4	8 GB
Large	Up to 250	8	16 GB
X-Large	Up to 500	16	32 GB
XX-Large	500+	32	64 GB

Figura 12: Tabla de requerimientos para la instalación de K3s  
Procedencia: <https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/>

La documentación de K3s aconseja también el uso de discos SSD, prioritariamente en los nodos maestros.

## 7. Viabilidad y riesgos

Por todo lo dicho hasta ahora, es evidente que está en la propia naturaleza del proyecto demostrar la viabilidad de la implementación del clúster en un entorno local de producción, es decir, un ámbito alejado de la práctica habitual de las empresas que contratan los servicios de Kubernetes en remoto a grandes plataformas, o de los entornos locales de pruebas.

Para convertir el proyecto en una realidad viable es necesario adaptar o sustituir soluciones diseñadas para los servicios en la nube por otros preparados para la infraestructura local. Por ejemplo, para crear un balanceador de carga o para desplegar un servicio de monitorización es necesario investigar programas de configuración manual y más compleja que la alternativa “natural”, como sería la simple contratación en una plataforma como AWS.

Kubernetes es software gratuito de código abierto, y por tanto el coste del procedimiento se limita a la adquisición, en su caso, o mantenimiento de la infraestructura física del sistema informático de la empresa. La dimensión de esta infraestructura dependerá de la carga de trabajo prevista en el clúster y del grado de observancia en la política de seguridad del sistema.

Tratándose de una instalación desde cero, también se minimizan los riesgos de tipo técnico.

Por otro lado, es interesante mencionar que el cumplimiento de objetivos descrito en la sección anterior es aplicable a un procedimiento de migración desde una arquitectura monolítica[11]. Y en este caso, no se pueden ignorar los riesgos inherentes a este tipo de operaciones, como la pérdida de datos o las incidencias de disponibilidad, con el consiguiente perjuicio para la empresa.

No obstante, demostrar un procedimiento de migración como el mencionado excede el objetivo de este proyecto.

## 8. Ejecución

Conforme a la planificación mostrada anteriormente, la fase de ejecución se estructurará en cuatro apartados:

1. Creación de una aplicación web funcional.
2. Creación del clúster de Kubernetes de alta disponibilidad.
  - 2.1. Crear las imágenes docker de la aplicación y subirlas al registro.
  - 2.2. K3s: creación de un clúster de Kubernetes de seis nodos.
3. Despliegue de las aplicaciones:
  - 3.1. Volúmenes persistentes de datos.
  - 3.2. Manifiestos.
4. Evaluación.
  - 4.1. Comprobación de la alta disponibilidad
  - 4.2. Comprobación del balanceo de carga
  - 4.3. Monitorización

## 8.1. Creación de una aplicación web funcional

La aplicación web que se implementará en el clúster de Kubernetes se diseña usando el patrón modelo-vista-controlador. La tarea que realiza consiste en disponer la información desde o hacia una base de datos. Consta de tres elementos:

1. La vista, *frontend*, o interfaz que se presenta al usuario, y que tiene por objeto exponer los datos de forma amigable y, en su caso, recogerlos a través del formulario.
2. El modelo: Una base de datos persistente donde se guarda la información con una estructura definida.
3. Una interfaz de programación de aplicaciones (API) que gestiona las peticiones de la vista hacia la base de datos, la devolución de información y captura los errores y los expone de forma más amigable al usuario.

Aunque la aplicación propuesta es sencilla, cuenta con funcionalidades suficientes para entender y comprobar el funcionamiento del clúster de Kubernetes de alta disponibilidad en infraestructura local. Por tanto, si bien no está preparada para ser desplegada en un entorno de producción, es válida para comprobar las funcionalidades básicas de dicho entorno.

En esta propuesta, la aplicación trata con datos referidos a los módulos y las unidades de un alumno. La información que aparece en la página principal de la web es una tabla con los módulos. En la cabecera de la página dos botones permiten seleccionar el contenido de la tabla entre módulos o unidades. En la parte superior de la tabla se dispone un formulario para la entrada o modificación de los datos.

La interacción con la tabla se produce cuando se pulsa sobre cualquiera de las filas, lo que dispara un evento que carga el registro correspondiente en el formulario. A partir de aquí se puede modificar el registro para su actualización en la base de datos. Por otro lado, al final de cada fila se muestra el icono de una papelera que permite borrar el registro de la fila donde se ubica.

<b>Id</b>	<b>Nombre</b>	<b>URL</b>	
ASGBD	Administración de Sistemas Gestores de Base de Datos	#	
ASO	Administración de Sistemas Operativos	#	

Figura 13: Detalle de dos filas en la aplicación

### 8.1.1. La vista

La interfaz web se implementa a partir de tres archivos:

- **index.html**: Contiene la estructura estática básica de la página web: encabezado, tabla y formulario. La estructura interna de la tabla, es decir, los elementos `<tr>` y `<td>`, se genera dinámicamente, por lo que no se incluye en este archivo.
- **style.css**: Contiene los estilos que se aplican al archivo HTML. Para la estructura del cuerpo principal de la página se usa la propiedad `grid` para crear un modelo sencillo de rejilla de dos columnas. Los elementos activos se resaltan en la interacción con el usuario con la pseudo-clase `:hover` y la propiedad `transition` para crear un efecto más estético.
- **script.js**: Contiene la lógica que gestiona la interacción entre el usuario y la interfaz web por un lado, y la información enviada a la API por otro.. En este archivo se configuran las peticiones de información al servidor. Para estas peticiones se usa la tecnología AJAX (Asynchronous JavaScript and XML), que permite la recuperación y exposición de los datos en pantalla sin necesidad de recargar el archivo HTML al completo. Para configurar el envío de la información hacia la API, la actividad de los eventos y la formación dinámica del código HTML se ha usado la librería de JavaScript llamada JQuery.

The screenshot shows a web-based application interface. At the top, there is a dark blue header bar with two buttons: "Módulos" and "Unidades". Below the header, there is a search form with fields for "Id" (containing a placeholder "00000000") and "Nombre" (containing a placeholder "Nombre"). To the right of these fields are buttons for "Limpiar" (Clear) and "Enviar" (Send). Below the search form is a table with columns "Id", "Nombre", and "URL". The table contains six rows of data:

Id	Nombre	URL
ASGBD	Administración de Sistemas Gestores de Base de Datos	# <input type="button" value="X"/>
ASO	Administración de Sistemas Operativos	# <input type="button" value="X"/>
EIE	Empresa e Iniciativa Emprendedora	# <input type="button" value="X"/>
IAW	Implantación de Aplicaciones Web	# <input type="button" value="X"/>
SAD	Seguridad y Alta Disponibilidad	# <input type="button" value="X"/>
SRI	Servicios de Red e Internet	# <input type="button" value="X"/>

Figura 14: Vista de la aplicación Web

### 8.1.2. La base de datos

Para la gestión de los datos de la aplicación se emplea una base de datos relacional MySQL llamada "modulos". La estructura, bastante sencilla, cuenta con una tabla que contiene la información de los módulos y otra de las unidades que contiene cada módulo:

- La tabla que contiene los módulos tiene tres campos: "id\_modulo", "modulo" y "url". El tipo de datos para todos ellos es VARCHAR y la clave primaria es el primer campo de los mencionados.
- La tabla que contiene las unidades tiene tres campos: "id\_modulo", "unidad" y "titulo".
  - Tiene una clave primaria compuesta por los dos primeros campos.
  - Además, "id\_modulo" es una clave foránea que se relaciona con el campo del mismo nombre de la tabla "modulos".
  - Respecto de los tipos de datos, todos son VARCHAR excepto el campo "unidad" que es TINYINT.

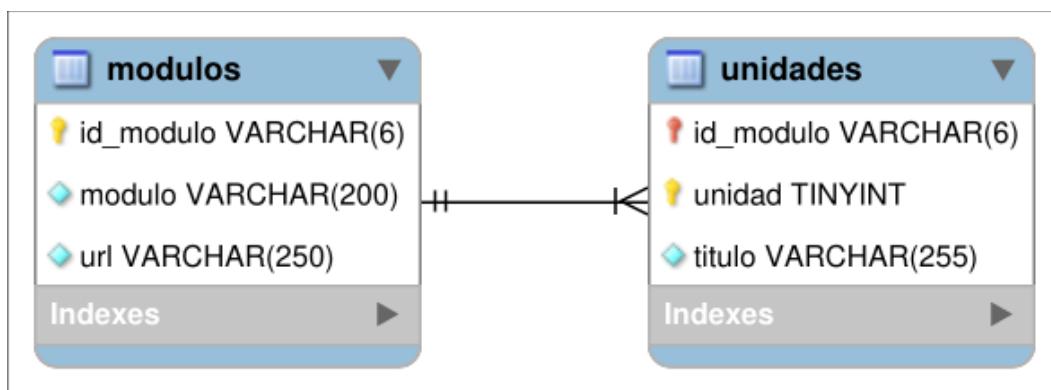


Figura 15: Esquema ER de la base de datos propuesta

### 8.1.3. API

Para crear la interfaz entre la web y la base de datos se ha usado Flask, un micro-framework para Python que facilita la elaboración de aplicaciones web con el patrón modelo-vista-controlador.

La API gestiona las peticiones HTTP hacia la base de datos y aplica determinada lógica de negocio en función de la ruta a la que se envía y del método de petición HTTP. Básicamente, relaciona cada una de las peticiones con la consulta apropiada a la base de datos.

Método HTTP	Ruta	Consulta	Acción
GET	/modulos	SELECT * FROM modulos;	Recupera todos los módulos
GET	/modulos/id	SELECT * FROM modulos where id_modulo="id";	Recupera el módulo con la clave primaria igual al valor de "id"
POST	/modulo	INSERT INTO modulos VALUES ("id_modulo", "modulo", "url");	Inserta un nuevo registro con los valores de un objeto con formato JSON que la API recibe desde el formulario web
PUT	/modulo/id	UPDATE modulos SET ("modulo"=id.modulo, "url"=id.url) where "id_modulo"=id.id_modulo)	Actualiza el registro cuya clave primaria coincide con el valor de la ruta, con los datos que se envían desde el formulario
DELETE	/modulo/id	DELETE modulos WHERE "id_modulo"=id	Elimina el registro de la tabla cuya clave primaria sea igual al valor de la cadena "id" en la ruta.

Tabla 3: Relación de peticiones HTTP y consultas MySQL en la tabla "modulo"

La transmisión de los datos entre la API y la base de datos es responsabilidad de una librería de Python llamada Flask-Mysqldb. Una vez recibe los datos, y aplicada la lógica que hayamos determinado, la API envía hacia el cliente web la información correspondiente en formato JSON.

```
~> curl localhost:5000/modulos/SRI
{
  "mensaje": "Modulo encontrado",
  "modulo": {
    "id": "SRI",
    "modulo": "Servicios de Red e Internet",
    "url": "#"
  }
}
```

Figura 16: Resultado de la petición GET de un módulo desde consola

## 8.2. Crear imágenes Docker de la aplicación para subirlas al registro

Cuando ejecutamos una aplicación en un contenedor Docker podemos descargar la imagen oficial desde Docker Hub y posteriormente realizar las modificaciones necesarias dentro del contenedor. Es decir, primero creamos el contenedor y posteriormente realizamos cambios. Como los cambios se realizan en el contenedor y no en la imagen, no son persistentes.

Otra opción consiste en crear una imagen con los cambios que deseemos, en vez de actuar sobre el contenedor. De esta manera, las modificaciones son persistentes: cada contenedor originado desde esa imagen replicará estos cambios.

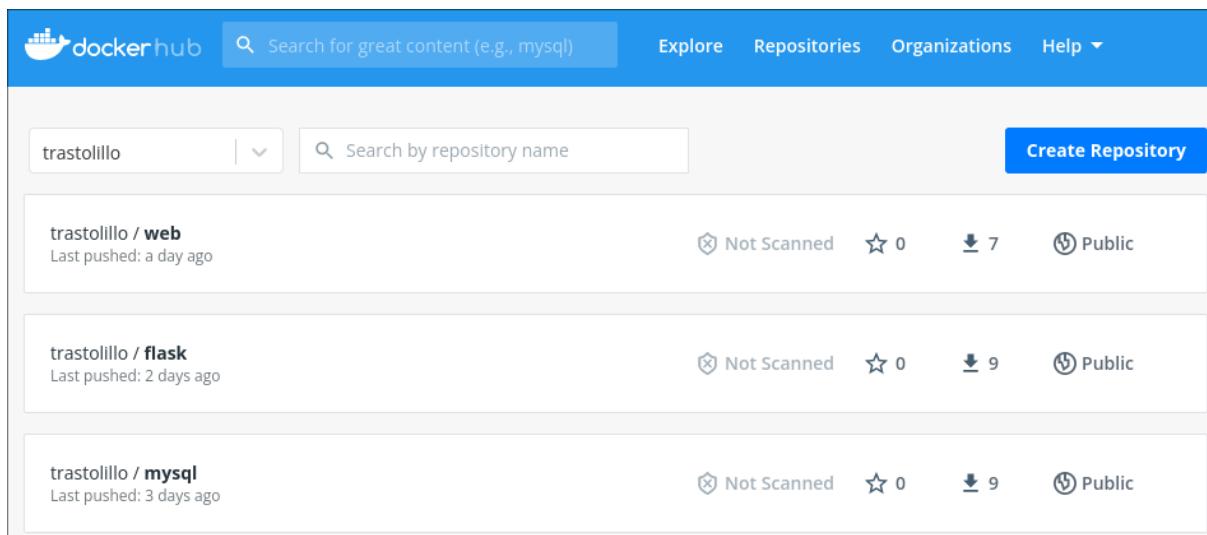


Figura 17: Captura del repositorio de Docker Hub

Este procedimiento se realiza en varios pasos:

1. Se crea un archivo, generalmente llamado Dockerfile, en el que determinamos la imagen desde la que se parte y los cambios a realizar.
2. Construimos la imagen modificada con el comando:

```
# docker build -t nombre-imagen -f Dockerfile /ubicacion-dockerfile
```

3. Si se ha construido correctamente aparecerá en el listado que devuelve "docker images". El siguiente paso consiste en etiquetar la imagen usando la dirección del registro, con el formato "dominio/imagen". Para ello se emplea el siguiente comando:

```
# docker tag nombre-de-la-imagen registro.com/nombre-de-la-imagen
```

4. Subimos la imagen al registro con el comando "docker push". Si lo hacemos a una cuenta privada en Docker Hub se debe sustituir el nombre de registro por el del usuario de la cuenta.:

```
# docker push registro/nombre-de-la-imagen
```

A continuación, se explicarán los Dockerfile que construyen las imágenes para los distintos componentes de la aplicación web.

#### Archivo: Dockerfile.mysql

```
FROM mysql:8.0
ENV LANG=C.UTF_8
COPY ./schema.sql /docker-entrypoint-initdb.d/:ro
```

Se trata de una construcción sencilla: desde una imagen oficial de MySQL versión 8.0 copiamos un script SQL que contiene las instrucciones para la creación de la base de datos y las tablas. Se incluye una variable de entorno para indicar el tipo de codificación de caracteres. Podrían incluirse otras variables de entorno relativas, por ejemplo, al nombre de la base de datos con la que se va a operar, el usuario o la contraseña de acceso al servidor MySQL. Sin embargo, para nuestro objetivo resulta más adecuado definir estas variables en los manifiestos de despliegue.

#### Archivo: Dockerfile.flask

```
FROM python:bullseye
WORKDIR /app
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV FLASK_ENV=development
ENV DEBIAN_FRONTEND=noninteractive
RUN apt update && apt install -y python3-mysqldb && rm -rf /var/lib/apt/lists/*
COPY ./backend/ .
RUN pip3 install -r requirements.txt
CMD ["flask", "run"]
```

Este archivo prepara una imagen para crear contenedores de la API en un entorno de pruebas. Como ya se ha dicho, no es apta para entornos de producción. No obstante, ilustra de forma bastante clara cómo se va construyendo una imagen en un Dockerfile.

En este caso se parte de una imagen de Debian 11. Seguidamente se define el directorio de trabajo, una forma de establecer una ubicación por defecto para las operaciones siguientes. Después, se pasan algunas variables de entorno, se instala el módulo de Python para MySQL y se copian los archivos del proyecto al directorio de trabajo. A partir del archivo "requirements.txt" se instalan todas las dependencias con el gestor de paquetes "pip". Finalmente, se ejecuta en la consola el comando "flask run" que pone en marcha el servidor.

#### Archivo: Dockerfile.nginx

```
FROM nginx:1.21
WORKDIR /usr/share/nginx/html
COPY ./html/ .
```

Para la vista de la aplicación se emplea la imagen de la versión 1.21 de nginx, copiando los archivos necesarios a la carpeta correspondiente al servidor web.

## 8.3. Creación de un clúster de Kubernetes de seis nodos

### 8.3.1. Planteamiento

La creación del clúster se realiza en tres etapas:

1. Creación del balanceador de carga de entrada al clúster.
2. Instalación de herramientas.
3. Creación de los nodos.

La ejecución se realizará en seis máquinas virtuales conectadas en la misma red local, cada una de ellas con 1 GB de memoria RAM, 15 GB de almacenamiento y con el sistema operativo Ubuntu Server 20.04 LTS.

Los roles se reparten de la siguiente manera:

- Equipos ajenos al clúster:
  - El equipo del administrador, desde el que se crea el clúster. Aquí contaremos con el software para la instalación: k3sup. En este caso empleamos el mismo equipo donde instalaremos el balanceador de carga. También instalaremos la herramienta de línea de comando kubectl para enviar comandos a la API del Kubernetes.
  - Balanceador de carga. Instalaremos un contenedor Docker con Nginx. Balancea la carga de las peticiones a la API en los servidores maestros del clúster.
- Nodos del clúster:
  - Tres nodos servidores o maestros.
  - Tres nodos de trabajo o workers.

### 8.3.2. Balanceador de carga

Conforme a la topografía propuesta, en primer lugar creamos el balanceador de carga que organiza las conexiones de entrada al clúster. Se trata de un equipo ajeno al clúster que recibirá las peticiones de acceso a sus componentes, en concreto a los nodos servidores o maestros.

Para obtener esta funcionalidad desplegamos un servidor nginx con el siguiente archivo docker-compose:

```
version: '3'
services:
  web:
    image: nginx
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - "6443:6443"
```

Se expone el puerto 6443, que debe estar accesible para la comunicación entre los nodos. También se comparte en un volumen persistente el archivo de configuración de nginx.conf, que determina la configuración del balanceo de carga entre los nodos maestros:

```
worker_processes 4;
worker_rlimit_nofile 40000;

events {
  worker_connections 8192;
}
stream {
  upstream k3snodes {
    server 192.168.122.10:6443 max_fails=3 fail_timeout=5s;
    server 192.168.120.20:6443 max_fails=3 fail_timeout=5s;
    server 192.168.122.30:6443 max_fails=3 fail_timeout=5s;
  }
  server {
    listen 6443;
    proxy_pass k3snodes;
  }
}
```

En este archivo podemos observar que el balanceo de carga se realiza entre las IP 192.168.122.10, 20 y 30, que son las correspondientes a los nodos maestros. El servidor

escucha en el puerto 6443, que recordemos que está vinculado al mismo puerto del equipo anfitrión. Finalmente, no se especifica ningún algoritmo de balanceo, por lo que empleará por defecto Round Robin.

### **8.3.3. Herramienta para instalación: K3sup**

K3sup[12] es una herramienta de código abierto que facilita la instalación del cluster de Kuberentes. Emplea SSH para instalar K3s en cada nodo. La instalación es un proceso automatizado que se realiza con el script de utilidad de Rancher, y que dispone de una opción para determinar la IP pública del host y que TLS funcione. Luego, como veremos, se obtiene el archivo "kubeconfig" en el servidor, actualizado para poder conectar con la API del clúster.

Figura 18: Instalación de K3sup (elaboración propia)

Encontramos las instrucciones de instalación de K3sup en su repositorio de GitHub. En nuestro caso, para economizar recursos instalaremos k3sup en el mismo equipo donde hemos instalado anteriormente el balanceador de carga de entrada al clúster. Para ello, ejecutamos con privilegios de administrador el siguiente comando:

```
curl -sLS https://get.k3sup.dev | sh
```

### 8.3.4. Preparación de las conexiones SSH

Para que K3sup pueda agregar los nodos al clúster, primero debemos configurar el acceso desatendido por SSH a cada nodo desde la máquina en que estemos operando para crear el clúster. Para ello, vamos a repetir los siguientes pasos:

1. Generamos la clave pública y privada desde la másquina en que creamos el clúster con el comando `ssh-keygen -t rsa`.
2. En cada nodo realizamos dos operaciones:

- 2.1. Creamos una contraseña para el usuario root si no lo tuviera:

```
$ sudo -i
# passwd
```

- 2.2. Configuramos el servidor SSH para que permite la conexión remota del root.

```
# sed -i 's/#PermitRootLogin prohibit-password/PermitRootLogin yes/g'
/etc/ssh/sshd_config
```

3. Creamos un archivo de configuración para las conexiones ssh en `~/.ssh/config`. Este paso no es necesario, pero cuando tenemos que acceder en remoto varias máquinas es bastante recomendable.

	<b>File: .ssh/config</b> Size: 595 B
1	<code>Host m1</code>
2	<code>Hostname 192.168.122.10</code>
3	<code>User root</code>
4	
5	<code>Host m2</code>
6	<code>Hostname 192.168.122.20</code>
7	<code>User root</code>
8	
9	<code>Host m3</code>
10	<code>Hostname 192.168.122.30</code>
11	<code>User root</code>
12	
13	<code>Host w1</code>
14	<code>Hostname 192.168.122.51</code>
15	<code>User root</code>
16	
17	<code>Host w2</code>
18	<code>Hostname 192.168.122.52</code>
19	<code>User root</code>

Figura 19: Archivo de configuración de alias SSH

4. Copiamos la clave pública en cada nodo con el comando `ssh-copy-id`. Por ejemplo, si tuviéramos alias para `m1`, `m2` y `m3` como los tres nodos máster, podríamos automatizar el procedimiento con el siguiente comando:

```
# for $i in {1..3}; do ssh-copy-id m$i; done
```

### 8.3.5. Creación del clúster: agregando el primer nodo

Finalizada la configuración del acceso SSH, creamos el clúster con un primer nodo maestro, que llamaremos "master1". Recordemos que la IP del equipo balanceador de carga es 192.168.122.1, y del primer nodo maestro es 192.168.1.10. Entonces, desde la máquina principal donde crearemos el clúster ejecutamos el siguiente comando de K3sup:

```
k3sup install \
--host=192.168.122.10 \
--user=root \
--cluster \
--ssh-key ~/.ssh/id_rsa \
--tls-san 192.168.122.1 \
--k3s-extra-args="\
--node-taint \
--disable servicelb \
--disable local-storage \
node-role.kubernetes.io/master=true:NoSchedule"
```

Los parámetros que se pasan al comando se explican así:

- **--host:** la IP en la LAN del nodo que vamos a crear.
- **--user:** el usuario con el que k3sup se conectará a través de SSH.
- **--ssh-key:** la clave del certificado que hemos empleado anteriormente para configurar el acceso SSH en modo desatendido.
- **--cluster:** Este parámetro indica a K3sup que la base de datos de gestión y estado del clúster será de tipo etcd y no será externa, sino integrada en el clúster.
- **--tls-san:** la IP del equipo que dispone de los certificados de autenticación para el acceso al clúster. En este caso es la del equipo que hace de balanceador de carga.
- **--k3s-extra-args:** Este parámetro permite pasar distintos argumentos. En este caso pasamos dos que definen el nodo como maestro.
  - **--disable servicelb:** Deshabilita el balanceador de carga integrado que usa K3s. Este parámetro es necesario para instalar MetalLB como balanceador de carga de nuestra elección.
  - **--disable local-storage:** Deshabilita el objeto "storage-class" por defecto del clúster. Instalaremos Longhorn como sistema de almacenaje persistente de datos distribuido en el clúster.

- Con "**--node-taint**" y "**NoSchedule**" indicamos que el nodo no acepta trabajos (en k3s, los nodos maestros pueden desplegar aplicaciones si no se indica lo contrario).

Hay que tener en cuenta que esta instrucción es única para crear el clúster, es decir, sólo se ejecuta una vez en el inicio del procedimiento. Más adelante, como veremos, se empleará "k3sup join" para ir agregando el resto de nodos, sean maestros o workers.

Una vez terminado el proceso de instalación se comprueba que K3sup ha creado un servicio en el nodo llamado k3s.service. Si no se ha producido ningún error, el servicio estará en ejecución.

```
Loaded: loaded (/etc/systemd/system/k3s.service; enabled; vendor preset: enabled)
Active: active (running) since Tue 2022-05-24 14:03:44 UTC; 6min ago
  Docs: https://k3s.io
Process: 4159 ExecStartPre=/bin/sh -xc ! /usr/bin/systemctl is-enabled --quiet nm-cloud-se>
Process: 4161 ExecStartPre=/sbin/modprobe br_netfilter (code=exited, status=0/SUCCESS)
Process: 4162 ExecStartPre=/sbin/modprobe overlay (code=exited, status=0/SUCCESS)
Main PID: 4163 (k3s-server)
  Tasks: 75
 Memory: 785.8M
 CGroup: /system.slice/k3s.service
         ├─4163 /usr/local/bin/k3s server
         ├─4174 containerd -c /var/lib/rancher/k3s/agent/etc/containerd/config.toml -a /ru>
         ├─4718 /var/lib/rancher/k3s/data/8c2b0191f6e36ec6f3cb68e2302fcc4be850c6db31ec5f8a>
         ├─4739 /var/lib/rancher/k3s/data/8c2b0191f6e36ec6f3cb68e2302fcc4be850c6db31ec5f8a>
         ├─4754 /pause
         ├─4787 /var/lib/rancher/k3s/data/8c2b0191f6e36ec6f3cb68e2302fcc4be850c6db31ec5f8a>
         ├─4815 /pause
         ├─4822 /pause
         ├─4885 /metrics-server --cert-dir=/tmp --secure-port=4443 --kubelet-preferred-add>
         ├─4968 local-path-provisioner start --config /etc/config/config.json
         ├─4998 /coredns -conf /etc/coredns/Corefile
```

Figura 20: Estado del servicio K3s en el nodo master1

### 8.3.6. Gestión del clúster: Instalación de kubectl

Kubectl es una herramienta de línea de comandos que accede a la API de Kubernetes para la gestión del clúster. Con ella podemos realizar todo tipo de operaciones, como obtener información sobre la estructura del clúster, desplegar aplicaciones, crear espacios de nombres, etc.

Hay que decir que K3sup instala kubectl en el servidor maestro que acabamos de instalar. Sin embargo, para que el acceso a la API se realice a través del balanceador de carga de los nodos maestros, conviene descargar e instalar esta herramienta en el equipo de administración del clúster. La instalación de kubectl se realiza en dos pasos:

1. Descarga del binario:

```
# curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
```

2. Instalación en el *path* para que se pueda ejecutar desde cualquier ubicación. Con permisos de administración se ejecuta el siguiente comando:

```
# install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

A continuación debemos configurar kubectl para que apunte al clúster. Generalmente, si se trata de un clúster en la nube, el proveedor pone a nuestra disposición un archivo llamado "Kubeconfig" que contiene la configuración para que kubectl obtenga la información del clúster correcto.

En nuestro caso, al crearse el primer nodo también se crea un archivo llamado "kubeconfig" en el directorio desde el que ejecutamos el comando "k3sup install". Este archivo contiene la información sobre certificados y dirección de red de entrada a la API del clúster. Lo moveremos al espacio del usuario en el directorio ".kube" y con el nombre "config". Así, Kubectl obtendrá automáticamente su configuración en esta ubicación. ejecutamos.

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0t
jTk1qSXd0VEkwTVRRd016QTRXaGN0TxpJd05USXhNVFF3TxpBNApXakFqTVNFd0h3WUP
LLV0VvVmNz0DhZR09SRDdvZmgYK0N3TEkKaTBpVTh3ZGfySThabUtScGJGczBjNmvdTy
1JKZTVWN0kzcHRSCjFTaEhiN2N3Q2dZSUtvWkl6ajBFQXjdJRFNRQXdSZ0loQuTzdEZBc
RU5EIENFU1RJRklDQVRFLS0tLS0K
    server: https://192.168.122.1:6443
  name: default
contexts:
- context:
    cluster: default
    user: default
  name: default
```

Figura 21: Resaltado de línea modificada en el archivo  
~/.kube/config

A continuación, tenemos que editar el archivo recién copiado. En concreto, modificaremos la línea que comienza con la palabra "server", que indica la IP del servidor que nos dará acceso a la API de Kubernetes. Como vamos a usar un balanceador de carga externo, colocaremos su IP: 192.168.122.1.

Podemos comprobar que kubectl conecta con el clúster recuperando la información sobre los nodos:

```
> kubectl get nodes
NAME      STATUS    ROLES          AGE      VERSION
master1   Ready     control-plane,etcd,master   16h      v1.23.6+k3s1
```

Figura 22: Información de los nodos del clúster con kubectl

### 8.3.7. Adición del resto de nodos

Para agregar el siguiente nodo maestro ejecutaremos la siguiente sentencia:

```
k3sup join \
--host=192.168.122.20 \
--server-user=root \
--server-host=192.168.122.10 \
--user=root \
--server
--k3s-extra-args="\
--node-taint \
--disable servicelb \
--disable local-storage \
node-role.kubernetes.io/master=true:NoSchedule"
```

Como se puede ver, ahora se debe ejecutar "k3sup join" en vez de "install", con las siguientes particularidades en el paso de los parámetros:

**--host:** La dirección de red del nodo que estamos agregando.

**--server-user:** El usuario en el primer nodo, que debe ser el mismo que colocamos en el parámetro "--user" del comando de instalación del clúster.

**--server-host:** Para el resto de nodos, se pondrá la dirección de red del primer nodo maestro que hemos agregado.

**--server:** Esta opción indica que el rol del equipo será el de servidor o maestro.

Agregaremos otro servidor maestro más, cambiando el parámetro correspondiente a su IP.

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	CONTAINER-RUNTIME
master1	Ready	control-plane,etcd,master	5h9m	v1.23.6+k3s1	192.168.122.10	<none>	Ubuntu 20.04.3 LTS	5.4.0-109-generic	containerd://1.5.11-k3s2
master2	Ready	control-plane,etcd,master	4h52m	v1.23.6+k3s1	192.168.122.20	<none>	Ubuntu 20.04.4 LTS	5.4.0-109-generic	containerd://1.5.11-k3s2
master3	Ready	control-plane,etcd,master	4h41m	v1.23.6+k3s1	192.168.122.30	<none>	Ubuntu 20.04.4 LTS	5.4.0-107-generic	containerd://1.5.11-k3s2
worker1	Ready	<none>	9m31s	v1.23.6+k3s1	192.168.122.31	<none>	Ubuntu 20.04.4 LTS	5.4.0-113-generic	containerd://1.5.11-k3s2
worker2	Ready	<none>	11s	v1.23.6+k3s1	192.168.122.32	<none>	Ubuntu 20.04.3 LTS	5.4.0-107-generic	containerd://1.5.11-k3s2
worker3	Ready	<none>	15s	v1.23.6+k3s1	192.168.122.33	<none>	Ubuntu 20.04.4 LTS	5.4.0-113-generic	containerd://1.5.11-k3s2

Figura 23: Lista de nodos del clúster ya completado

A continuación agregamos los nodos agentes o workers. Para ello utilizamos el mismo comando "k3sup join", pero sin la opción "--server" y eliminando los argumentos opcionales.

```
k3sup join \
--host=192.168.122.51 \
--server-user=root \
--server-host=192.168.100.51 --user=root
```

## 8.4. Instalación de un balanceador interno: MetalLB

Como ya se explicó, los servicios pueden ser externos o internos. Un servicio externo de tipo LoadBalancer expone una IP pública, o varias. En la siguiente figura podemos observar cómo Kubernetes asigna las IP de todos los nodos en funcionamiento a la API de nuestra aplicación. Este tipo de comportamiento se soluciona con la instalación de MetalLB.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
flask-service	LoadBalancer	10.43.178.70	192.168.122.10,192.168.122.20,192.168.122.30,192.168.122.31,192.168.122.32,192.168.122.33

Figura 24: Un servicio tipo LoadBalancer con sus IP externas

MetalLB[13] es un balanceador de carga para instalaciones de Kubernetes en entornos locales que propone una solución diferente a lo expuesto para la asignación de IP externas. Cuando configuramos este programa reservamos para él un grupo de IP que irá asignando a los servicios externos según se vayan creando, balanceando la carga desde esa IP asignada a los distintos nodos de la aplicación.

Este programa dispone de dos modos para realizar el balanceo de carga: un modo en la capa 2, mediante ARP, y otro modo en la capa 7, mediante el protocolo BGP. En nuestro caso se implementará con el primer tipo de balanceo de carga.

kubectl -n metallb-system get all						
NAME	READY	STATUS	RESTARTS	AGE		
pod/controller-57fd9c5bb-ppkbv	1/1	Running	0	10m		
pod/speaker-6x5dj	1/1	Running	0	10m		
pod/speaker-b8ws7	1/1	Running	0	10m		
pod/speaker-jpqdl	1/1	Running	0	10m		
pod/speaker-r8wth	1/1	Running	0	10m		
pod/speaker-t5qpx	1/1	Running	0	10m		
pod/speaker-xwg7k	1/1	Running	0	10m		
NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR AGE
daemonset.apps/speaker	6	6	6	6	6	kubernetes.io/os=linux 10m
NAME	READY	UP-TO-DATE	AVAILABLE	AGE		
deployment.apps/controller	1/1	1	1	10m		
NAME	DESIRED	CURRENT	READY	AGE		
replicaset.apps/controller-57fd9c5bb	1	1	1	10m		

Figura 25: Salida en consola de todos los objetos del namespace de MetalLB

La instalación se realiza en tres pasos:

1. Se crea el espacio de nombres:

```
$ kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.12.1/manifests/
namespace.yaml
```

2. Aplicamos el manifiesto que instalará MetalLB en el clúster:

```
$ kubectl apply -f  
https://raw.githubusercontent.com/metallb/metallb/v0.12.1/manifests/metallb.yaml
```

3. Se aplica el manifiesto de tipo ConfigMap para determinar el rango de direcciones IP que va a usar MetalLB para identificar los servicios expuestos. En este caso reservamos el rango 192.168.122.230-192.168.122.250:

```
12 lines (12 sloc) | 221 Bytes  
  
1  apiVersion: v1  
2  kind: ConfigMap  
3  metadata:  
4    namespace: metallb-system  
5    name: config  
6  data:  
7    config: |  
8      address-pools:  
9        - name: default  
10       protocol: layer2  
11       addresses:  
12         - 192.168.122.230-192.168.122.250
```

Figura 26: ConfigMap para configurar MetalLB

## 8.5. Despliegue de aplicaciones

Una vez establecida la forma principal del clúster con tres servidores administradores y tres nodos de trabajo, continuamos declarando el resto de la estructura.

Como ya se dijo, una característica básica de los Pods es que son entidades efímeras. Por otro lado, cualquier dato almacenado en el Pod desaparece cuando el orquestador lo suprime. En consecuencia, para satisfacer la necesidad de mantener los datos de forma persistente tenemos que crear volúmenes independientes del ciclo de vida los Pods. En nuestro caso crearemos un volumen persistente para los datos de la base de datos de la aplicación.

Finalmente, se realizará el despliegue de cada componente de la aplicación web mediante manifiestos. Los manifiestos son archivos en formato yaml que mediante sintaxis declarativa que determina el estado en el que debe conformarse la estructura dentro de clúster. Los usaremos para la definición de los volúmenes persistentes, para guardar los datos sensibles en los llamados "secretos", para los despliegues de la aplicación y los distintos servicios, y para determinar la forma de ingreso del usuario a la aplicación.

### 8.5.1. Volumen persistente de datos

Kubernetes no ofrece persistencia de datos por defecto. Es el administrador del clúster el que tendrá que activar esta característica. Los volúmenes persistentes de datos conservan la información generada por la aplicación independientemente del Pod donde se haya originado.

Ya hemos determinado que se trata de independizar los volúmenes de datos del ciclo de vida del Pod. Pero además, deben ser independientes de la actividad del propio clúster, de forma que incluso en el caso de que todo el clúster caiga la información debe conservarse.

Hay que entender que el soporte material, real, del almacenamiento es responsabilidad de la persona que administra el clúster, y por tanto, el componente "**PersistentVolume**" de Kubernetes únicamente ofrece una interfaz para que el administrador disponga ese almacenamiento en el clúster. Pero es este administrador el que debe decidir qué tipo de almacenamiento y su configuración, de las copias de seguridad, etc.

En resumen, para que un volumen de almacenamiento corresponda con un clúster de alta disponibilidad tiene que cumplir dos requisitos:

1. No estar vinculado en exclusiva a ningún nodo.
2. Mantenerse si el clúster deja de funcionar.

En la siguiente figura podemos observar el ejemplo de un manifiesto para declarar un volumen de 2 Gi en un servidor NFS:

```
17 lines (17 sloc) | 289 Bytes

1  apiVersion: v1
2  kind: PersistentVolume
3
4  metadata:
5    name: mysql-pv-volume
6    labels:
7      type: local
8      author: Gabriel
9
10 spec:
11   storageClassName: manual
12   capacity:
13     storage: 2Gi
14   accessModes:
15     - ReadWriteMany
16
17   nfs:
18     path: /mnt/data/
19     server: 192.168.122.5
20     readOnly: false
```

**Tipo de componente y versión de la API**

**Metadatos del componente:**  
- Nombre del volumen persistente.  
- Etiquetas sobre el tipo de volumen (local) y el administrador que lo configura

**Características del almacenamiento:**  
- Capacidad.  
- Forma de acceso.

**Tipo de almacenamiento. Servidor nfs:**  
- path: Ruta donde se va a guardar  
- server: dirección de red del servidor

Figura 27: Manifiesto para un volumen persistente de tipo nfs

Ahora bien, objeto *PersistentVolume* es un componente pasivo, a la espera de las aplicaciones que lo reclamen. Para que las aplicaciones y programas en el clúster soliciten lugar dentro de ese espacio, deben hacer uso de otro componente: **PersistentVolumeClaim**.

```
11 lines (11 sloc) | 190 Bytes

1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4    name: mysql-pv-claim
5  spec:
6    storageClassName: manual
7    accessModes:
8      - ReadWriteOnce
9    resources:
10   requests:
11     storage: 2Gi
```

Figura 28: Manifiesto para un PVC

Se observa en la última línea de la imagen anterior que este manifiesto reclama 2 Gi del volumen persistente recién creado.

Esta sería la manera de realizar una implementación sencilla para disponer de un espacio donde guardar los datos. No obstante, implementaremos otra solución distinta, especialmente diseñada para clúster de kubernetes en *bare-metal* o en local: Longhorn.

### 8.5.1.1. Longhorn

Existe otro objeto relacionado con el almacenamiento que supone una capa de abstracción sobre los volúmenes persistentes y los manifiestos de solicitud de espacio: la clase de almacenamiento o **StorageClass**. Como observamos en el manifiesto de ejemplo de la figura anterior, reclamamos espacio para una aplicación mysql y agregamos el atributo storageClassName para determinar la clase de almacenamiento. K3s integra un StorageClass llamado "local-storage".

Ahora bien, en nuestro caso hemos deshabilitamos este StorageClass por defecto en el momento de creación del clúster para posibilitar la instalación de Longhorn que se describe continuación.

Según aparece en su documentación, Longhorn[14] es "un sistema de almacenamiento de bloques distribuido ligero, confiable y fácil de usar para Kubernetes". Permite guardar datos en los nodos del clúster de forma distribuida, replicar los volúmenes de almacenamiento y agregar nuevas unidades de almacenamiento donde crear nuevos volúmenes. Además, como se verá, dispone de una interfaz web de usuario que monitoriza el espacio y permite asignar volúmenes en el clúster. Es por tanto, una herramienta versátil que soluciona el problema del almacenamiento persistente.

Longhorn exige una serie de requisitos para su instalación: versiones modernas de Kuberentes, containerd, un sistema de archivos ext4 o XFS, determinadas herramientas de bash como curl, grep y awk entre otras.

Nuestros servidores con Ubuntu 20.04 cumplen con todos los requisitos excepto uno: debemos instalar en cada nodo que deseemos emplear para el almacenamiento un cliente NFSv4. Como en este caso sólo vamos a destinar a tal uso los nodos workers, para instalar en todos el paquete "nfs-common" usamos un pequeño script, descrito en los anexos, que ejecuta en cada nodo el comando:

```
# apt install nfs-common -y
```

Con los requisitos ya satisfechos, vamos a instalar Longhorn en el clúster usando el gestor de paquetes "helm":

1. Agregamos el repositorio de Loghorn:

```
$ helm repo add longhorn https://charts.longhorn.io && \
  helm repo update
```

2. Antes de instalarlo, guardamos los valores que usa el manifiesto en un archivo llamado `longhorn-values.yaml`.

```
$ helm show values longhorn/longhorn > ./longhorn-values.yaml
```

3. Ahora sí, instalamos Longhorn. Creará su propio espacio de nombre, llamado "longhorn-system" y desplegará todos los objetos en él:

```
$ helm install longhorn longhorn/longhorn --namespace longhorn-system \
--create-namespace
```

Longhorn ya está instalado, pero si observamos los servicios comprobamos que todos son de tipo ClusterIP, es decir, internos. Por tanto, a continuación expondremos el servicio encargado de dar acceso a los Pods que ejecutan el panel de control de Longhorn.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
csi-attacher	ClusterIP	10.43.189.19	<none>	12345/TCP	36s
csi-provisioner	ClusterIP	10.43.84.5	<none>	12345/TCP	35s
csi-resizer	ClusterIP	10.43.97.7	<none>	12345/TCP	31s
csi-snapshotter	ClusterIP	10.43.9.55	<none>	12345/TCP	31s
longhorn-backend	ClusterIP	10.43.49.254	<none>	9500/TCP	2m8s
longhorn-engine-manager	ClusterIP	None	<none>	<none>	2m9s
longhorn-frontend	ClusterIP	10.43.120.129	<none>	80/TCP	2m8s
longhorn-replica-manager	ClusterIP	None	<none>	<none>	2m9s

Figura 29: El servicio `longhorn-frontend` se encarga de la interfaz gráfica de Longhorn

Para ello recuperamos el manifiesto de valores que creamos antes de realizar la instalación y cambiamos el atributo correspondiente al tipo de servicio, de ClusterIP a LoadBalancer:

```
46   service:
47     ui:
48       type: LoadBalancer
49     nodePort: null
50   manager:
51     type: ClusterIP
52     nodePort: ""
53
```

Figura 30: El servicio responsable de la interfaz cambiado a LoadBalancer

Y, finalmente, actualizamos Longhorn con el cambio:

```
$ helm upgrade --install longhorn longhorn/longhorn --values longhorn-
values.yaml -n longhorn-system
```

Si listamos los servicios nuevamente, podemos comprobar que el balanceador de carga del clúster, MetalLB, le ha asignado una de las direcciones del rango que establecimos en el punto anterior, 192.168.122.231, y lo ha mapeado al puerto del nodo 30235.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
csi-attacher	ClusterIP	10.43.189.19	<none>	12345/TCP	11m
csi-provisioner	ClusterIP	10.43.84.5	<none>	12345/TCP	11m
csi-resizer	ClusterIP	10.43.97.7	<none>	12345/TCP	11m
csi-snapshotter	ClusterIP	10.43.9.55	<none>	12345/TCP	11m
longhorn-backend	ClusterIP	10.43.49.254	<none>	9500/TCP	13m
longhorn-engine-manager	ClusterIP	None	<none>	<none>	13m
longhorn-frontend	LoadBalancer	10.43.120.129	192.168.122.231	80:30235/TCP	13m
longhorn-replica-manager	ClusterIP	None	<none>	<none>	13m

Figura 31: El servicio longhorn-frontend con IP externa

Ahora podemos acceder a la interfaz gráfica de Longhorn a través de la dirección 192.168.122.231. MetalLB balanceará la carga a los nodos designados para el almacenamiento por el puerto 30235.

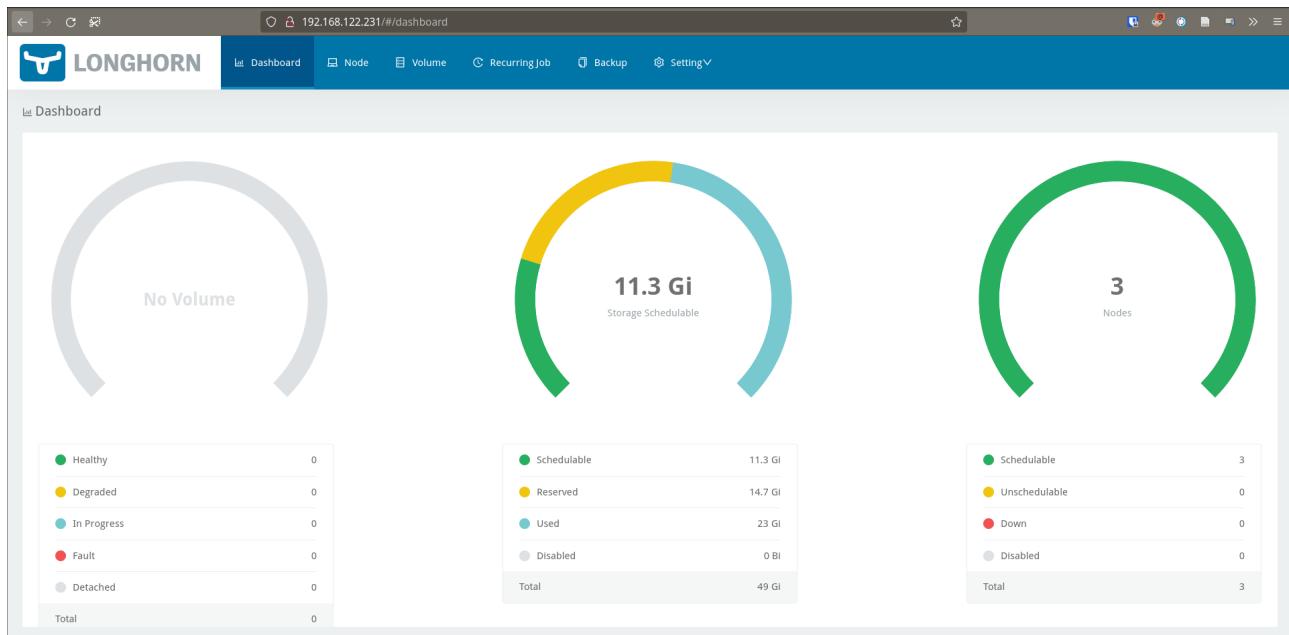


Figura 32: Interfaz gráfica web de Longhorn

Más adelante, cuando creamos el volumen de datos para la aplicación, observaremos más detalles de esta interfaz gráfica.

### **8.5.2. Despliegue de la base de datos**

La base de datos es el único componente de la aplicación que requiere un volumen de datos persistente.

Por tanto, desplegaremos el gestor MySQL en tres etapas:

1. Desplegaremos los manifiestos que disponen la contraseña del servidor de base de datos y la URL interna para emplear desde la API.
2. Crearemos un volumen de datos persistente (PersistentVolume) y le asociaremos una solicitud espacio (PersistentVolumeClaim). Esto lo haremos a través de la interfaz de Longhorn.
3. Por último, se desarrollará un manifiesto para desplegar la base de datos (tipo "Deployment") y su correspondiente servicio.

### 8.5.2.1. Secret y ConfigMap

En primer lugar creamos un componente llamado "Secret".

```
7 lines (7 sloc) | 103 Bytes
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: mysql-secret
5  type: Opaque
6  data:
7    password: cGFzc3dvcmQ=
```

Figura 33: Secreto de la base de datos

Se detalla en su manifiesto el tipo, que será "Opaque", es decir, no se mostrará la información en texto plano sino codificada con base64, y finalmente la contraseña, para ser usada en otros manifiestos sin necesidad de exponerla, como se verá a continuación.

Seguidamente, aplicamos un manifiesto tipo ConfigMap donde se especifica la URL de la base de datos dentro del espacio de nombres donde se despliega, que debe coincidir con el nombre del servicio que dará acceso a la base de datos.

```
6 lines (6 sloc) | 100 Bytes
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: mysql-configmap
5  data:
6    database_url: mysql-service
```

Figura 34: ConfigMap de la base de datos

### 8.5.2.2. Creación de volúmenes de datos para MySQL

Desde la interfaz web de Longhorn creamos el volumen de 2 Gi con dos réplicas, como se ve en la siguiente imagen.

Create Volume

\* Name: mysql-pv

\* Size: 2 Gi

\* Number of Replicas: 2

\* Frontend: Block Device

Data Locality: disabled

Access Mode: ReadWriteMany

Backing Image:

Replicas Auto Balance: ignored

Disable Revision Counter:

Encrypted:

Node Tag:

Disk Tag:

Cancel OK

Figura 35: Menú de creación de un PV en Longhorn

Seguidamente creamos la solicitud de espacio (PVC) vinculándolo al volumen recién creado.

Volume

Delete Attach Detach Create Backup

	State	Name	Size	Actual Size
<input checked="" type="checkbox"/>	Detached	mysql-pv	2 Gi	0 Bi

Create PV/PVC

\* PV Name: mysql-pv

Access Mode: ReadWriteMany

File System: Ext4 XFS

Create PVC:  Use Previous PVC:

\* PVC Name: mysql-pvc

\* Namespace: default

Custom Column Go

Name	Attached To	Schedule	Last Backup At	Operation

Attached To Schedule Last Backup At Operation

Cancel OK

Figura 36: Menú de creación de un PVC en Longhorn

Una vez creados ambos objetos, comprobamos en la interfaz que el volumen aparece como vinculado a un PVC.

The screenshot shows a table in the Longhorn interface with the following columns: State, Name, Size, Actual Size, Created, PV/PVC, and Namespace. A red box highlights the 'PV/PVC' column for the row where the status is 'Bound'. The table data is as follows:

	State	Name	Size	Actual Size	Created	PV/PVC	Namespace
<input type="checkbox"/>	Detached	mysql-pvc	2 Gi	0 Bi	3 minutes ago	Bound	default

Figura 37: Registro de un volumen vinculado en la interfaz de Longhorn

Asimismo, Podemos observar el manifiesto del PVC creado por Longhorn, solicitando en la consola la salida de información en formato yaml.

```
> kubectl get pvc mysql-pvc -o yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    pv.kubernetes.io/bind-completed: "yes"
  creationTimestamp: "2022-06-02T22:02:18Z"
  finalizers:
  - kubernetes.io/pvc-protection
  name: mysql-pvc
  namespace: default
  resourceVersion: "22011"
  uid: 3de26cc9-a160-48b9-b919-6586317aa2b5
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
  storageClassName: longhorn-static
  volumeMode: Filesystem
  volumeName: mysql-pv
status:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 2Gi
  phase: Bound
```

Figura 38: Archivo yaml generado por Longhorn

### 8.5.2.3. Aplicación del manifiesto

Ahora podemos crear el manifiesto de despliegue de la base de datos. Como suele ocurrir con este tipo de manifiestos consta de dos partes: en la parte superior se declara el despliegue en si mismo, y en la parte inferior se anexa el manifiesto del servicio asociado a esa aplicación.

Veamos en primer lugar el manifiesto de despliegue en el archivo mysql-deployment.yaml:

De este manifiesto podemos destacar algunos detalles:

- El nombre del manifiesto es "mysql-deployment".
- Creará un Pod en el clúster que albergará un contenedor con la imagen del servidor MySQL descargada de un repositorio privado de Docker Hub.
- Se pasa una variable de entorno, "MYSQL\_ROOT\_PASSWORD", para determinar la contraseña de acceso al servidor MySQL. Sin embargo, el valor de esa variable no se muestra el manifiesto, sino que lo toma del secreto creado anteriormente.
- En el manifiesto se hace uso de la solicitud de espacio que hemos creado con Longhorn.

Cerrando el archivo, creamos el manifiesto para el servicio que dará acceso a la base de datos:

```
36  ---
37  apiVersion: v1
38  kind: Service
39  metadata:
40    name: mysql-service
41  spec:
42    ports:
43      - protocol: TCP
44        port: 3306
45        targetPort: 3306
46    selector:
47      app: mysql
```

Figura 39: Manifiesto para el servicio MySQL

Aquí podemos observar lo siguiente:

- El tipo de servicio es ClusterIP. Como este tipo es el que se aplica por defecto no es necesario declararlo.

- El atributo "port" se refiere al puerto del Pod, mientras que el atributo targetPort es el puerto que abre el contenedor. En ambos se mantiene el puerto por defecto del servidor MySQL, 3306.
- El selector "app: mysql" debe coincidir con la etiqueta correspondiente en el manifiesto de despliegue mostrado anteriormente.

#### 8.5.2.4. Un apunte sobre MySQL y Kubernetes

El despliegue realizado consta de un Pod de MySQL que dispone de un espacio de almacenamiento en dos réplicas.

The screenshot shows the Longhorn interface with a volume named 'mysql-pv'. On the left, under 'Volume Details', the state is 'Attached', health is 'Healthy', and it's ready for workload. It has two replicas: 'worker1' and 'worker3'. Both replicas are 'pv-r-474fc9ee' and are labeled as 'Healthy'. They are running on 'instance-manager-r-01a0acbd' and 'instance-manager-r-5997e8a1' respectively, with paths '/var/lib/longhorn/' and '/var/lib/longhorn/'. The size of the volume is 2 Gi and data locality is disabled.

Replica	Path	Status
worker1	/var/lib/longhorn/	Running
worker3	/var/lib/longhorn/	Running

Figura 40: Detalle del volumen para MySQL en la interfaz de Longhorn

Por un lado, el orquestador siempre va a procurar que exista al menos una instancia de MySQL, de forma que si el nodo donde está desplegado deja de funcionar lo recrearía en otro. Y por otro lado, de igual manera disponemos de dos volúmenes de datos en dos distintos nodos. Podríamos crear más réplicas para los datos, o incluso agregar una unidad de almacenamiento externa o un servidor NFS. Es decir, este planteamiento satisface la exigencia de un sistema de alta disponibilidad.

Debemos tener en cuenta que las réplicas de almacenamiento se realizan en el ámbito del sistema de archivos, mientras que las réplicas de la base de datos correspondiente es responsabilidad del gestor de MySQL y pertenece al ámbito de la aplicación y su configuración.

Dicho esto, en la documentación de Kubernetes se expone un ejemplo de despliegue para MySQL tal vez más apropiado para un entorno de producción de alta disponibilidad[15]. Se trata de ejecutar un manifiesto de tipo StatefulSet, es decir, para desplegar una aplicación con estado, como una base de datos. Este despliegue constará de varias réplicas: una principal con permisos de escritura y lectura, y el resto con permisos de sólo lectura.

Para diferenciar los permisos se crea un ConfigMap con dos configuraciones, e igualmente dos servicios.

Finalmente, cabe mencionar que podemos encontrar en la documentación de diversos servicios en la nube la valoración sobre la conveniencia de implementar una base de datos re-

lacional en Kubernetes, sus beneficios y desventajas, en términos de eficiencia y del tratamiento de los datos[16].

### 8.5.3. Despliegue de la API

Creamos el documento de despliegue de la API como se ve en la siguiente imagen.

```
48 lines (47 sloc) | 958 Bytes

1  apiversion: apps/v1
2  kind: Deployment
3  metadata:
4    name: flaskapi-deployment
5    labels:
6      app: flaskapi
7  spec:
8    replicas: 2
9    selector:
10   matchLabels:
11     app: flaskapi
12   template:
13     metadata:
14       labels:
15         app: flaskapi
16     spec:
17       containers:
18         - name: flaskapi
19           image: trastolillo/flask
20           imagePullPolicy: IfNotPresent
21         ports:
22           - containerPort: 5000
23         env:
24           - name: password-mysql
25             valueFrom:
26               secretKeyRef:
27                 name: mysql-secret
28                 key: password
29           - name: host-mysql
30             valueFrom:
31               configMapKeyRef:
32                 name: mysql-configmap
33                 key: database_url
```

Figura 41: Manifiesto de despliegue de la API

Se puede observar que las variables de entorno relativas a la contraseña y la ruta de la base de datos se obtienen de las claves correspondientes dentro del secreto y el config-map, respectivamente, que hemos creado antes.

También destaca la política de descargar la imagen desde el repositorio cuando no exista en local (atributo "imagePullPolicy") y la exposición del puerto 5000 del contenedor.

A continuación en el mismo archivo declaramos el servicio que dará acceso a los Pod de la API. Vemos que es de tipo externo, LoadBalancer. Por tanto, al exponerse para el acceso desde el exterior del Pod, al puerto del pod y del contenedor se añade un nuevo atributo: nodePort, el puerto del nodo. El número de este puerto debe estar en el rango comprendido entre 31000 a 32767.

```

36   apiVersion: v1
37   kind: Service
38   metadata:
39     name: flask-service
40   spec:
41     ports:
42       - port: 5000
43         protocol: TCP
44         targetPort: 5000
45         nodePort: 31000
46     selector:
47       app: flaskapi
48     type: LoadBalancer

```

Figura 42: Manifiesto para el servicio de la API

Después de aplicar el archivo podemos observar como MetalLb ha asignado al servicio la siguiente IP en el conjunto que habíamos declarado como assignable a servicios externos.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	
flask-service	LoadBalancer	10.43.211.254	192.168.122.232	5000:31000/TCP	28m	
kubernetes	ClusterIP	10.43.0.1	<none>	443/TCP	20h	
mysql-service	ClusterIP	10.43.220.44	<none>	3306/TCP	28m	

Figura 43: Listado de servicios del espacio de nombres por defecto

Finalmente, podemos comprobar desde la consola que la API conecta correctamente con la base de datos ejecutando una llamada con "curl".

```

~> curl 192.168.122.232:5000/modulos/SRI
{
  "mensaje": "Modulo encontrado",
  "modulo": {
    "id": "SRI",
    "modulo": "Servicios de Red e Internet",
    "url": "#"
  }
}

```

Figura 44: Llamada a la API en la consola

### 8.5.4. Despliegue de la vista de la aplicación

Para el despliegue de la vista de la web creamos el manifiesto de la siguiente imagen.

```
41 lines (41 sloc) | 696 Bytes

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: vista-deployment
5  spec:
6    selector:
7      matchLabels:
8        run: vista
9    replicas: 3
10   template:
11     metadata:
12       labels:
13         run: vista
14     spec:
15       containers:
16         - name: vista
17           image: trastolillo/web
18           imagePullPolicy: IfNotPresent
19         ports:
20           - containerPort: 80
21       resources:
22         requests:
23           memory: "300Mi"
24           cpu: "500m"
25         limits:
26           memory: "500Mi"
27           cpu: "700m"

resources:
  requests:
    memory: "300Mi"
    cpu: "500m"
  limits:
    memory: "500Mi"
    cpu: "700m"
```

Se determinan los recursos.  
 La unidad de memoria es  
 el mibibyte (MiB), la del cpu es  
 el milicore.

Figura 45: Manifiesto para el despliegue de la página web

Como aspecto novedoso, en este archivo hemos determinado un límite de recursos en la sección "resources" del manifiesto. Se establecen los que se asignan al contenedor y también un límite máximo. Si se supera este límite el orquestador destruirá el Pod y lo recreará nuevamente.

NAME	READY	STATUS	RESTARTS	AGE
flaskapi-deployment-5c5796c755-f8cd2	1/1	Running	0	151m
flaskapi-deployment-5c5796c755-tntgb	1/1	Running	0	132m
mysql-deployment-f879fbdbf-l67vk	1/1	Running	0	144m
vista-deployment-5647c8859b-4c7q9	1/1	Running	0	52s
vista-deployment-5647c8859b-hr975	1/1	Running	0	52s
vista-deployment-5647c8859b-sgb9p	1/1	Running	0	52s

Figura 46: Todos los Pod de la aplicación web

El servicio que dará acceso a la vista es de tipo interno. Para acceder a la página web creamos dos nuevos objetos: IngressController e Ingress.

```
29  apiVersion: v1
30  kind: Service
31  metadata:
32    name: vista-svc
33    labels:
34      run: vista
35  spec:
36    ports:
37      - port: 80
38        targetPort: 80
39        protocol: TCP
40    selector:
41      run: vista
```

*Figura 47: Manifiesto para el servicio de la vista Web*

### 8.5.5. Acceso a la web: Nginx Ingress Controller

El Ingress y el IngressController son objetos de API de Kubernetes.

Ingress expone las rutas HTTP y HTTPS desde fuera del clúster a los servicios dentro del clúster. El enrutamiento del tráfico está controlado por reglas definidas en el recurso Ingress, como veremos.

El Ingress no es suficiente para el acceso externo a la web. Es preciso otro objeto, un Ingress Controller, que gestione las reglas de los distintos Ingress que pueden existir en el clúster. En los clúster en la nube el Ingress Controller está gestionado por el proveedor y es transparente al usuario. Para implementarlo en el entorno local, y siempre conforme a la documentación de NGINX Ingress Controller[17], aplicamos un archivo yaml con varios manifiestos que se instalarán en un espacio de nombre llamado "ingress-nginx".

```
$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.2.0/
deploy/static/provider/cloud/deploy.yaml
```

Una vez aplicado, si listamos los servicios del espacio de nombres de NGINX Ingress Controller, observamos que uno de ellos dispone de IP externa.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
ingress-nginx-controller	LoadBalancer	10.43.116.2	192.168.122.233	80:31955/TCP,443:31421/TCP	13m
ingress-nginx-controller-admission	ClusterIP	10.43.155.24	<none>	443/TCP	13m

Figura 48: Servicios de NGINX Ingress Controller

Esta será la IP hacia la que enviaremos el tráfico que entre al balanceador de carga externo con la URL de nuestra página web. Pero antes debemos crear el manifiesto Ingress para nuestra aplicación con la regla correspondiente.

```
17 lines (17 sloc) | 308 Bytes
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: web-ingress
5  spec:
6    rules:
7      - host: web.home
8        http:
9          paths:
10            - path: /
11              pathType: Prefix
12              backend:
13                service:
14                  name: vista-svc
15                  port:
16                    number: 80
17      ingressClassName: nginx
```

Figura 49: Manifiesto Ingress

El manifiesto contiene las siguientes reglas:

- La regla del Ingress establece que la URL de acceso será web.home por el puerto 80.
- El servicio al que se accede a través de esta regla es "vista-svc", creado en el punto anterior.
- El tipo de IngressClass es nginx. De forma análoga a lo que sucedía con el StorageClass, el IngressClass es una capa de abstracción sobre el Ingress Controller. Es este atributo el que vincula el manifiesto Ingress con Nginx Ingress Controller.

Después de aplicado, si lo listamos, la salida en consola ofrece información sobre la URL y sobre la IP de acceso, que coincide con la que vimos en el servicio de Ingress-Controller.

```
kubectl get ingress
NAME      CLASS   HOSTS          ADDRESS        PORTS   AGE
web-ingress  nginx  web.home    192.168.122.233  80      44s
```

Figura 50: Información sobre el Ingress de la aplicación

Una vez añadido el registro DNS para que "web.home" apunte a la IP 192.168.122.233, ya podemos acceder a la aplicación web con el navegador introduciendo la dirección.

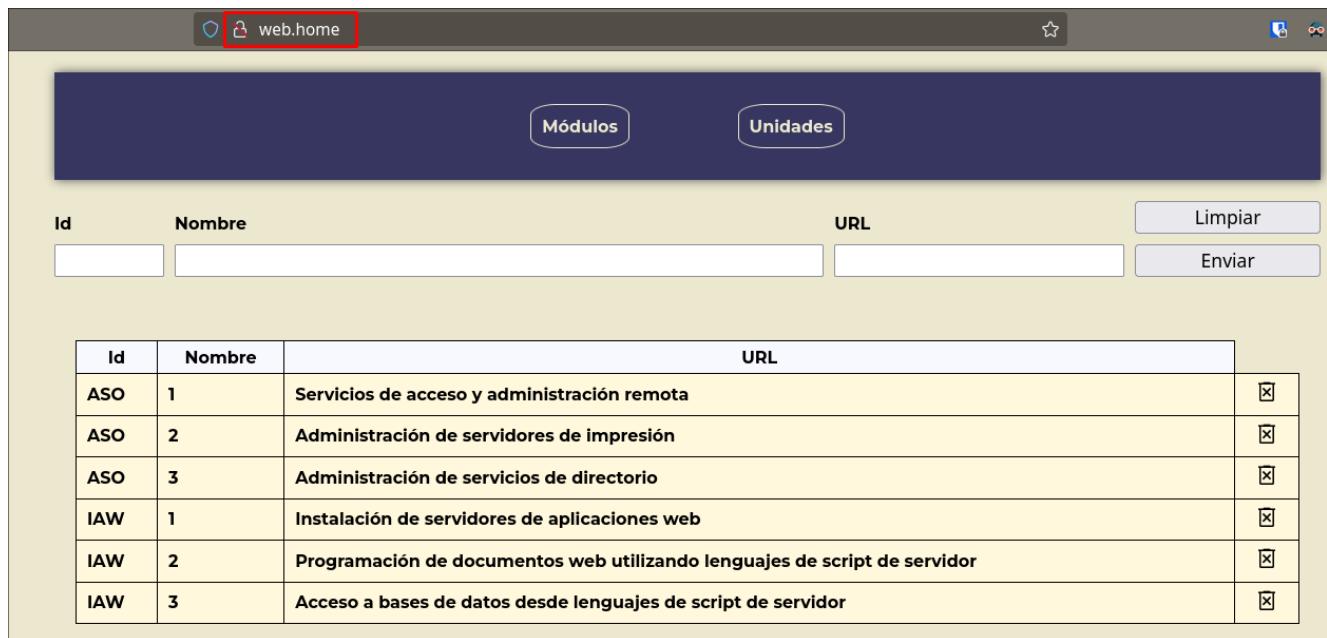


Figura 51: Acceso a la aplicación Web desplegada en el clúster

Después de comprobar que conecta con la API correctamente, se da por finalizado el despliegue de la aplicación.

## 9. Evaluación

La evaluación se realizará desde distintas perspectivas:

1. Comprobación de que la aplicación web es operativa, es decir, que el entorno de producción se desempeña correctamente.
2. Comprobación de que las características más ventajosas de Kubernetes (escalabilidad, alta disponibilidad y balanceo de carga) se aplican correctamente.
3. Monitorización de los procesos de Kubernetes mediante el despliegue en el clúster de software especializado en mediciones de rendimiento y control de los nodos.

## 9.1. Operatividad de la aplicación

La aplicación debe disponer desde el primer momento los datos de la base de datos y permitir realizar las operaciones de recuperación, modificación, agregación y borrado de registros.

Comprobamos en primer lugar la recuperación: seleccionando cualquier registro se hace una llamada a la API con la clave del registro y la información de ese registro en la base de datos se muestra en los campos de la parte superior.

The screenshot shows a web browser window titled "web.home". At the top, there are two buttons: "Módulos" and "Unidades". Below the buttons is a table with columns "Id", "Nombre", and "URL". A row is selected, showing "EIE" in the Id column and "Empresa e Iniciativa Emprendedora" in the Nombre column. To the right of the table are two buttons: "Limpiar" and "Enviar". Below the table is a larger table with columns "Id", "Nombre", and "URL". This table lists several records: ASGBD, ASO, EIE, IAW, SAD, and SRI. The "EIE" row is highlighted with a gray background, indicating it is the selected record. The "Nombre" column for EIE contains the text "Empresa e Iniciativa Emprendedora". The "URL" column for EIE contains the text "#". The "Nombre" column for ISO contains the text "Implantación de Sistemas Operativos". The "URL" column for ISO contains the text "www.iso.com".

Id	Nombre	URL
EIE	Empresa e Iniciativa Emprendedora	#
ASGBD	Administración de Sistemas Gestores de Base de Datos	#
ASO	Administración de Sistemas Operativos	#
EIE	Empresa e Iniciativa Emprendedora	#
IAW	Implantación de Aplicaciones Web	#
SAD	Seguridad y Alta Disponibilidad	#
SRI	Servicios de Red e Internet	#

Figura 52: Aplicación web recuperando un registro de la base de datos

Igualmente, se comprueba que se puede agregar un nuevo registro sin problemas.

The screenshot shows a web browser window titled "web.home". At the top, there are two buttons: "Módulos" and "Unidades". Below the buttons is a table with columns "Id", "Nombre", and "URL". The "Nombre" column is empty. To the right of the table are two buttons: "Limpiar" and "Enviar". Below the table is a larger table with columns "Id", "Nombre", and "URL". This table lists several records: ASGBD, ASO, EIE, IAW, ISO, SAD, and SRI. The "ISO" row is highlighted with a gray background, indicating it is the selected record. The "Nombre" column for ISO contains the text "Implantación de Sistemas Operativos". The "URL" column for ISO contains the text "www.iso.com".

Id	Nombre	URL
ASGBD	Administración de Sistemas Gestores de Base de Datos	#
ASO	Administración de Sistemas Operativos	#
EIE	Empresa e Iniciativa Emprendedora	#
IAW	Implantación de Aplicaciones Web	#
ISO	Implantación de Sistemas Operativos	www.iso.com
SAD	Seguridad y Alta Disponibilidad	#
SRI	Servicios de Red e Internet	#

Figura 53: Registro insertado a través de la aplicación web

Probamos a recuperar el nuevo registro y a modificarlo.

The screenshot shows a web-based application interface. At the top, there is a header bar with a logo and the text "web.home". Below the header, there are two buttons: "Módulos" and "Unidades". The main area contains a search form with fields for "Id" (containing "ISO") and "Nombre" (containing "Implantación de Sistemas Operativos - Con modificaciones"). There are also fields for "URL" (containing "www.iso.com") and buttons for "Limpiar" and "Enviar". Below the search form is a table with columns "Id", "Nombre", and "URL". The table rows are:

Id	Nombre	URL
ASGBD	Administración de Sistemas Gestores de Base de Datos	# <input checked="" type="checkbox"/>
ASO	Administración de Sistemas Operativos	# <input checked="" type="checkbox"/>
EIE	Empresa e Iniciativa Emprendedora	# <input checked="" type="checkbox"/>
IAW	Implantación de Aplicaciones Web	# <input checked="" type="checkbox"/>
ISO	Implantación de Sistemas Operativos	www.iso.com <input checked="" type="checkbox"/>
SAD	Seguridad y Alta Disponibilidad	# <input checked="" type="checkbox"/>
SRI	Servicios de Red e Internet	# <input checked="" type="checkbox"/>

Figura 54: Recuperación de un registro para su modificación

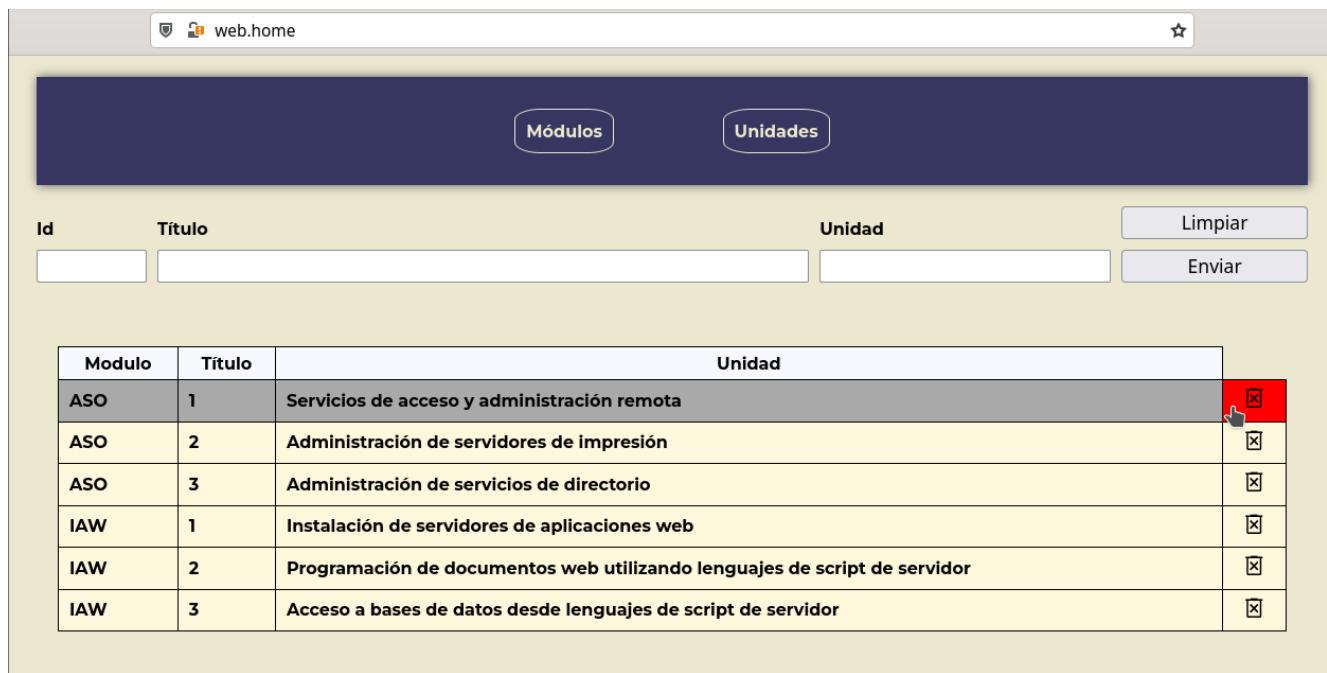
La tabla se actualiza mostrando que la modificación en la base de datos se ha realizado correctamente.

This screenshot shows the same web application interface as Figure 54, but with a different set of data in the table. The table rows are:

Id	Nombre	URL
ASGBD	Administración de Sistemas Gestores de Base de Datos	# <input checked="" type="checkbox"/>
ASO	Administración de Sistemas Operativos	# <input checked="" type="checkbox"/>
EIE	Empresa e Iniciativa Emprendedora	# <input checked="" type="checkbox"/>
IAW	Implantación de Aplicaciones Web	# <input checked="" type="checkbox"/>
ISO	Implantación de Sistemas Operativos - Con modificaciones	www.iso.com <input checked="" type="checkbox"/>
SAD	Seguridad y Alta Disponibilidad	# <input checked="" type="checkbox"/>
SRI	Servicios de Red e Internet	# <input checked="" type="checkbox"/>

Figura 55: Tabla con el registro modificado

Finalmente comprobamos que se los registros se pueden borrar. Pulsamos sobre el icono de eliminación del registro a descartar, en este caso de la tabla de unidades.



The screenshot shows a web-based application interface. At the top, there are two buttons: 'Módulos' and 'Unidades'. Below this is a search bar with fields for 'Id', 'Título', and 'Unidad', and buttons for 'Limpiar' (Clear) and 'Enviar' (Send). A table lists various units, each with a delete icon (a red square with a white 'X') in the last column. The first row, which contains the text 'Servicios de acceso y administración remota', has its delete icon highlighted with a red box and a cursor pointing at it.

Modulo	Título	Unidad
ASO	1	Servicios de acceso y administración remota
ASO	2	Administración de servidores de impresión
ASO	3	Administración de servicios de directorio
IAW	1	Instalación de servidores de aplicaciones web
IAW	2	Programación de documentos web utilizando lenguajes de script de servidor
IAW	3	Acceso a bases de datos desde lenguajes de script de servidor

Figura 56: Selección de un registro para eliminar

Y comprobamos que el registro se ha eliminado correctamente de la base de datos.



This screenshot shows the same web application after one record has been deleted. The table now displays only five rows of data, starting from 'ASO 2' and ending with 'IAW 3'. The delete icons are still present in the last column of each row.

Modulo	Título	Unidad
ASO	2	Administración de servidores de impresión
ASO	3	Administración de servicios de directorio
IAW	1	Instalación de servidores de aplicaciones web
IAW	2	Programación de documentos web utilizando lenguajes de script de servidor
IAW	3	Acceso a bases de datos desde lenguajes de script de servidor

Figura 57: Tabla con un registro ya eliminado

Se comprueba, por tanto, la operatividad de la aplicación web.

## 9.2. Comprobación de características de Kubernetes

### 9.2.1. Escalabilidad

La escalabilidad es una de las ventajas más características del clúster. La escalabilidad vertical viene facilitada por la flexibilidad que ofrece el orquestador para, por ejemplo, distribuir entre el resto de nodos los recursos de un nodo desconectado para su ampliación.

Sin embargo, Kubernetes destaca en la escalabilidad horizontal, añadiendo y quitando Pods a los distintos componentes de la aplicación web sin detener ningún recurso del clúster.

Partimos del número ya conocido de réplicas para componente de la aplicación. En este caso, vamos a centrar la atención en la API que dispone de dos réplicas: una de ellas en el nodo worker2 y otra en el worker3.

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
flaskapi-deployment-5c5796c755-gtmxt	1/1	Running	1 (6m25s ago)	6h32m	10.42.5.77	worker3
flaskapi-deployment-5c5796c755-hr26d	1/1	Running	1 (6m24s ago)	6h32m	10.42.4.76	worker2
mysql-deployment-675f577d6c-qc2x4	1/1	Running	0	71s	10.42.5.87	worker3
vista-deployment-5647c8859b-5nlwk	1/1	Running	1 (6m25s ago)	6h31m	10.42.5.81	worker3
vista-deployment-5647c8859b-6s4vm	1/1	Running	1 (6m24s ago)	6h31m	10.42.4.75	worker2
vista-deployment-5647c8859b-wcw49	1/1	Running	1 (6m25s ago)	6h31m	10.42.3.70	worker1

Figura 58: Réplicas de la API

Podemos escalar el despliegue de dos maneras:

1. Desde el manifiesto yaml ajustando el número de réplicas y aplicándolo nuevamente con "kubectl apply -f archivo.yaml"
2. Actualizando en caliente, desde la línea de comandos. Por ejemplo,

Para comprobar la escalabilidad al instante, vamos a suponer por un aumento de la demanda de peticiones de usuarios debemos añadir manualmente dos Pods más para la API empleando este segundo método:

```
$ kubectl scale --replicas=4 deployment flaskapi-deployment
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
Flaskapi-deployment-5c5796c755-g5lk8	1/1	Running	0	3m31s	10.42.5.88	worker3
Flaskapi-deployment-5c5796c755-gtmxt	1/1	Running	1 (19m ago)	6h45m	10.42.5.77	worker3
Flaskapi-deployment-5c5796c755-hr26d	1/1	Running	1 (19m ago)	6h45m	10.42.4.76	worker2
Flaskapi-deployment-5c5796c755-w9fbr	1/1	Running	0	3m31s	10.42.3.84	worker1
mysql-deployment-675f577d6c-qc2x4	1/1	Running	0	14m	10.42.5.87	worker3
vista-deployment-5647c8859b-5nlwk	1/1	Running	1 (19m ago)	6h44m	10.42.5.81	worker3
vista-deployment-5647c8859b-6s4vm	1/1	Running	1 (19m ago)	6h44m	10.42.4.75	worker2
vista-deployment-5647c8859b-wcw49	1/1	Running	1 (19m ago)	6h44m	10.42.3.70	worker1

Figura 59: Aumento de réplicas de la API

Si obtenemos información sobre el despliegue ejecutando el comando "kubectl describe" obtenemos el log de eventos del recurso flaskapi-deployment, donde también comprobamos que el escalado se realiza correctamente.

Type	Status	Reason			
Progressing	True	NewReplicaSetAvailable			
Available	True	MinimumReplicasAvailable			
OldReplicaSets: <none>					
NewReplicaSet: flaskapi-deployment-5c5796c755 (4/4 replicas created)					
Events:	Type	Reason	Age	From	Message
					-----
Normal	ScalingReplicaSet	6h53m	deployment-controller	Scaled up replica set flaskapi-deployment-5c5796c755 to 2	
Normal	ScalingReplicaSet	11m	deployment-controller	Scaled up replica set flaskapi-deployment-5c5796c755 to 4	

Figura 60: Salida de "kubectl describe deployment flaskapi-deployment"

Como hemos visto, es muy sencillo escalar un despliegue de forma manual. No obstante, lo habitual es programar el escalamiento en función de determinados parámetros. Por ejemplo, vamos a determinar que en caso de poca demanda, el número de réplicas para la vista de la aplicación web descienda de las actuales tres a una. Ejecutamos el siguiente comando:

```
$ kubectl autoscale deployment vista-deployment --cpu-percent=10 --min=1 --max=5
```

Esta línea establece que existirá un mínimo de un pod y un máximo de cinco para mantener una utilización promedio del diez por ciento de uso de CPU. Recordemos que en el manifiesto de la vista de la web habíamos reservado un uso de 500 milicos para el componentes, con un máximo de 700.

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
vista-deployment	Deployment/vista-deployment	0%/10%	1	5	1	4m5s

Figura 61: Información sobre la escalabilidad horizontal del despliegue

Para obtener la información sobre el estado automático de la escalabilidad de un despliegue, debemos listar el elemento "horizontalpodautoscaler" ejecutando en consola "kubectl get hpa". En el log de eventos del despliegue observamos que el número de réplicas ha autoescalado correctamente a una.

Type	Status	Reason			
Progressing	True	NewReplicaSetAvailable			
Available	True	MinimumReplicasAvailable			
OldReplicaSets: <none>					
NewReplicaSet: vista-deployment-5647c8859b (1/1 replicas created)					
Events:	Type	Reason	Age	From	Message
					-----
Normal	ScalingReplicaSet	7h17m	deployment-controller	Scaled up replica set vista-deployment-5647c8859b to 3	
Normal	ScalingReplicaSet	2m53s	deployment-controller	Scaled down replica set vista-deployment-5647c8859b to 1	

Figura 62: Log de eventos de vista-deployment

## 9.2.2. Alta disponibilidad

En el contexto del clúster de Kubernetes, el concepto de alta disponibilidad significa que la infraestructura debe mantenerse funcional independientemente de los eventos que puedan ocurrir. El administrador del clúster establece en la definición de la estructura los parámetros mínimos con los que se define la operatividad del clúster. Los componentes de Kubernetes llamados `kube-scheduler` y `kube-controller-manager`, de los que ya hablamos., son los que hacen posible la adecuación de la actividad del clúster a estos requisitos mínimos de funcionamiento.

Vamos a poner a prueba la disponibilidad del clúster apagando algunos nodos. Partimos de una situación inicial de seis nodos, tres maestros y tres workers.

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP
master1	Ready	control-plane,etcd,master	2d20h	v1.23.6+k3s1	192.168.122.10
master2	Ready	control-plane,etcd,master	2d20h	v1.23.6+k3s1	192.168.122.20
master3	Ready	control-plane,etcd,master	2d20h	v1.23.6+k3s1	192.168.122.30
worker1	Ready	<none>	2d20h	v1.23.6+k3s1	192.168.122.31
worker2	Ready	<none>	2d20h	v1.23.6+k3s1	192.168.122.32
worker3	Ready	<none>	2d20h	v1.23.6+k3s1	192.168.122.33

Figura 63: Número de nodos antes de la prueba de alta disponibilidad

Y por otro lado, disponemos de seis Pods repartidos entre los tres nodos workers.

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
flaskapi-deployment-5c5796c755-hr26d	1/1	Running	1 (6h22m ago)	12h	10.42.4.76	worker2
flaskapi-deployment-5c5796c755-w9fbr	1/1	Running	0	6h5m	10.42.3.84	worker1
mysql-deployment-675f577d6c-qc2x4	1/1	Running	0	6h16m	10.42.5.87	worker3
vista-deployment-5647c8859b-44ttj	1/1	Running	0	10s	10.42.4.82	worker2
vista-deployment-5647c8859b-wcw49	1/1	Running	1 (6h22m ago)	12h	10.42.3.70	worker1
vista-deployment-5647c8859b-x6f4p	1/1	Running	0	10s	10.42.5.89	worker3

Figura 64: Número de Pods antes de la prueba de alta disponibilidad

Probamos a apagar el nodo worker2.

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP
master1	Ready	control-plane,etcd,master	2d21h	v1.23.6+k3s1	192.168.122.10
master2	Ready	control-plane,etcd,master	2d21h	v1.23.6+k3s1	192.168.122.20
master3	Ready	control-plane,etcd,master	2d21h	v1.23.6+k3s1	192.168.122.30
worker1	Ready	<none>	2d21h	v1.23.6+k3s1	192.168.122.31
worker2	NotReady	<none>	2d21h	v1.23.6+k3s1	192.168.122.32
worker3	Ready	<none>	2d21h	v1.23.6+k3s1	192.168.122.33

Figura 65: Estado del clúster al apagar el nodo worker2

En unos instantes Kubernetes advierte que el nodo está caído y comienza a reprogramar los Pods en otros nodos.

En la siguiente imagen se observa en amarillo los Pod en proceso de desactivación por su ubicación en el nodo caído y en verde los Pod recreados en su sustitución en un nodo activo. En todo momento la página web se ha mantenido funcionando en su solicitudes a la base de datos.

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
flaskapi-deployment-5c5796c755-6tx7g	1/1	Running	0	52s	10.42.5.96	worker3
flaskapi-deployment-5c5796c755-hr26d	1/1	Terminating	1 (6h31m ago)	12h	10.42.4.76	worker2
flaskapi-deployment-5c5796c755-w9fbr	1/1	Running	0	6h15m	10.42.3.84	worker1
mysql-deployment-675f577d6c-qc2x4	1/1	Running	1 (78s ago)	6h26m	10.42.5.87	worker3
vista-deployment-5647c8859b-44ttj	1/1	Terminating	0	10m	10.42.4.82	worker2
vista-deployment-5647c8859b-bmf9x	1/1	Running	0	53s	10.42.5.94	worker3
vista-deployment-5647c8859b-wcw49	1/1	Running	1 (6h31m ago)	12h	10.42.3.70	worker1
vista-deployment-5647c8859b-x6f4p	1/1	Running	0	10m	10.42.5.89	worker3

Figura 66: Reprogramación de Pods por caída de un nodo

Apagamos ahora otro nodo: el nodo worker3. En una estructura en producción esto supondría una situación de emergencia importante, la caída de dos tercera partes de la infraestructura con la carga de trabajo de la aplicación.

NAME	STATUS	ROLES	AGE	VERSION
master1	Ready	control-plane,etcd,master	2d21h	v1.23.6+k3s1
master2	Ready	control-plane,etcd,master	2d21h	v1.23.6+k3s1
master3	Ready	control-plane,etcd,master	2d21h	v1.23.6+k3s1
worker1	Ready	<none>	2d21h	v1.23.6+k3s1
worker2	NotReady	<none>	2d21h	v1.23.6+k3s1
worker3	NotReady	<none>	2d21h	v1.23.6+k3s1

Figura 67: El clúster funcionando con un único nodo worker

Si listamos los recursos "deployment" obtenemos una idea más clara del desastre, con dos despliegues funcionando con un sólo pod y otro, la base de datos, sin ningún Pod.

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
flaskapi-deployment	1/2	2	1	13h
mysql-deployment	0/1	1	0	6h48m
vista-deployment	1/3	3	1	13h

Figura 68: Estado de los despliegues tras la caída de nodos

Es obvio que debido a que sólo disponemos de una réplica de la base de datos, por razones ya mencionadas, esto significa un fallo de las exigencias de un sistema de alta disponibilidad. Sin embargo, en aproximadamente dos minutos el sistema se recupera.

kubectl get deployments				
NAME	READY	UP-TO-DATE	AVAILABLE	AGE
flaskapi-deployment	2/2	2	2	13h
mysql-deployment	1/1	1	1	7h12m
vista-deployment	3/3	3	3	13h

Figura 69: Estado de los despliegues ya recuperados

Y si listamos los Pods, podemos observar que se han reprogramado todos los Pods en el único nodo worker que queda activo. En la siguiente imagen se muestra en verde los nuevos Pods y en amarillo los destruidos por la caída del nodo.

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
flaskapi-deployment-5c5796c755-6tx7g	1/1	Terminating	0	24m	10.42.5.96	worker3
flaskapi-deployment-5c5796c755-hr26d	1/1	Terminating	1 (6h55m ago)	13h	10.42.4.76	worker2
flaskapi-deployment-5c5796c755-kw24w	1/1	Running	0	23s	10.42.3.92	worker1
flaskapi-deployment-5c5796c755-w9fbr	1/1	Running	0	6h39m	10.42.3.84	worker1
mysql-deployment-675f577d6c-l6sng	1/1	Running	0	23s	10.42.3.98	worker1
mysql-deployment-675f577d6c-qc2x4	1/1	Terminating	1 (24m ago)	6h50m	10.42.5.87	worker3
vista-deployment-5647c8859b-44ttj	1/1	Terminating	0	33m	10.42.4.82	worker2
vista-deployment-5647c8859b-7lsls	1/1	Running	0	23s	10.42.3.89	worker1
vista-deployment-5647c8859b-bmf9x	1/1	Terminating	0	24m	10.42.5.94	worker3
vista-deployment-5647c8859b-vj4tf	1/1	Running	0	23s	10.42.3.97	worker1
vista-deployment-5647c8859b-wcw49	1/1	Running	1 (6h55m ago)	13h	10.42.3.70	worker1
vista-deployment-5647c8859b-x6f4p	1/1	Terminating	0	33m	10.42.5.89	worker3

Figura 70: Reprogramación de Pods tras la caída del segundo nodo

En resumen, hemos ilustrado un caso extremo en el caen dos de las tres máquinas ocupadas en la aplicación. El caso habitual será la eliminación de un Pod concreto por parte del orquestador y tal vez su recreación en otro, porque la demanda de usuarios desciende o porque el Pod ha superado los recursos reservados para su funcionamiento, por ejemplo. En estos casos, la sustitución del pod es prácticamente instantánea.

### 9.2.3. Balanceo de carga

Para probar el balanceo de carga vamos a modificar la página Web desde cada pod para diferenciar cada réplica. Cada HTML tendrá solamente una etiqueta h1 centrada con el nombre del nodo en el que se aloja. De esta manera sabremos qué servidor no está respondiendo.

Para acceder a la shell de un Pod se ejecuta el siguiente comando:

```
$ kubectl exec -ti [nombre-del-pod] -- bash
```

Para demostrar el balanceo de carga escribiremos un pequeño bucle en la línea de comandos, como sigue:

```
$ for i in {1..15}; do echo '-----'; curl web.home; sleep 1s; done
```

De esta forma, hacemos quince llamadas a la url de la aplicación web, una cada segundo, y podremos observar cómo se balancea la carga y varía el nodo que atiende la petición y devuelve una respuesta. En la siguiente imagen podemos apreciarlo.

```

~ ) for i in {1..15}; do echo '-----'; curl web.home; sleep 1s; done
-----
<h1>NODO 2</h1>
-----
<h1>NODO 3</h1>
-----
<h1>NODO 1</h1>
-----
<h1>NODO 3</h1>
-----
<h1>NODO 1</h1>
-----
<h1>NODO 2</h1>
-----
<h1>NODO 2</h1>
-----
<h1>NODO 3</h1>
-----
<h1>NODO 3</h1>
-----
<h1>NODO 1</h1>
-----
<h1>NODO 1</h1>
-----
<h1>NODO 2</h1>
-----
<h1>NODO 2</h1>
-----
<h1>NODO 3</h1>
-----
<h1>NODO 3</h1>

```

Figura 71: Demostración del balanceo de carga de clúster

## 9.3. Monitorización

### 9.3.1. Prometheus y Grafana

Prometheus[18] es un software de monitorización y alertas que almacena datos y métricas en una base de datos como series temporales, es decir, junto al instante de tiempo en que se ha registrado. Además, dispone de un servidor HTTP que acepta peticiones. Este componente web es usado por Grafana[19], que es una interfaz de usuario para la observación de los datos recogidos por Prometheus.

Para instalar Prometheus con Grafana en el clúster de Kubernetes usaremos el repositorio de kube-prometheus[20]. Conforme a la documentación del repositorio, debemos descargar o clonar la carpeta "manifests" del repositorio. Una vez ubicados en ella instalaremos el software ejecutando los siguientes comandos:

```
$ creating the remaining resources
kubectl apply --server-side -f manifests/setup
until kubectl get servicemonitors --all-namespaces ; do date; sleep 1; echo "";
done
kubectl apply -f manifests/
```

El programa se instala en el *namespace* llamado "monitoring". Si listamos los servicios de este espacio de nombres observamos un servicio para Grafana:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
alertmanager-main	ClusterIP	10.43.49.76	<none>	9093/TCP,8080/TCP	76s
alertmanager-operated	ClusterIP	None	<none>	9093/TCP,9094/TCP,9094/UDP	36s
blackbox-exporter	ClusterIP	10.43.80.246	<none>	9115/TCP,19115/TCP	75s
grafana	ClusterIP	10.43.149.218	<none>	3000/TCP	72s
kube-state-metrics	ClusterIP	None	<none>	8443/TCP,9443/TCP	70s
node-exporter	ClusterIP	None	<none>	9100/TCP	69s
prometheus-adapter	ClusterIP	10.43.22.244	<none>	443/TCP	62s
prometheus-k8s	ClusterIP	10.43.165.188	<none>	9090/TCP,8080/TCP	64s
prometheus-operated	ClusterIP	None	<none>	9090/TCP	32s
prometheus-operator	ClusterIP	None	<none>	8443/TCP	59s

Figura 72: Servicios del espacio de nombres "monitoring"

Una forma de acceder a la interfaz de usuario de Grafana es hacer un mapeo de puertos de su servicio al puerto 3000 nuestro equipo y entrar por el navegador en localhost:3000.

```
) kubectl --namespace monitoring port-forward svc/grafana 3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
```

Figura 73: Mapeo de puertos para entrar en la UI de Grafana

Para hacer el mapeo de puertos ejecutamos el siguiente comando en consola:

```
$ kubectl --namespace monitoring port-forward svc/grafana 3000
```

La primera vez que accedemos a Grafana las credenciales son admin/admin, pero nos obliga a establecer una nueva contraseña.

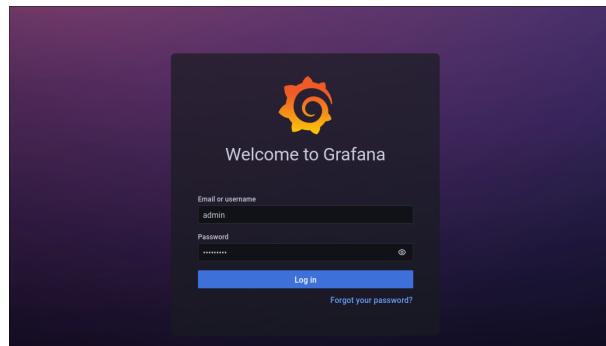


Figura 74: Página de acceso de Grafana

Una vez en la interfaz de Grafana debemos agregar una fuente de datos, que será Prometheus.

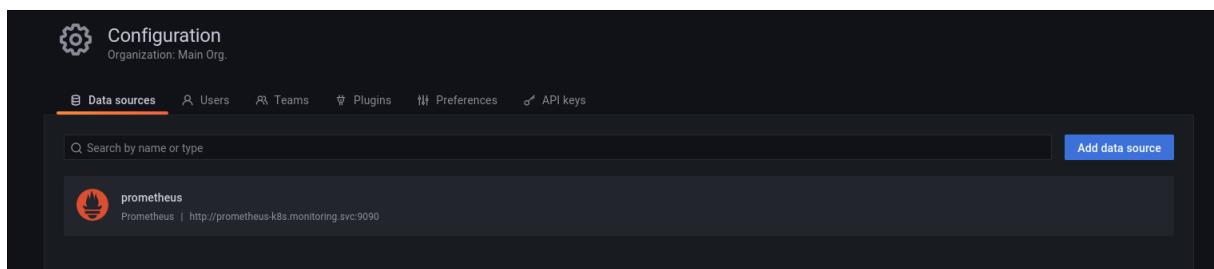


Figura 75: Sección para agregar una fuente de datos en Grafana

Seguidamente, seleccionamos los paneles informativos que deseamos mostrar. Seleccionaremos también Prometheus.

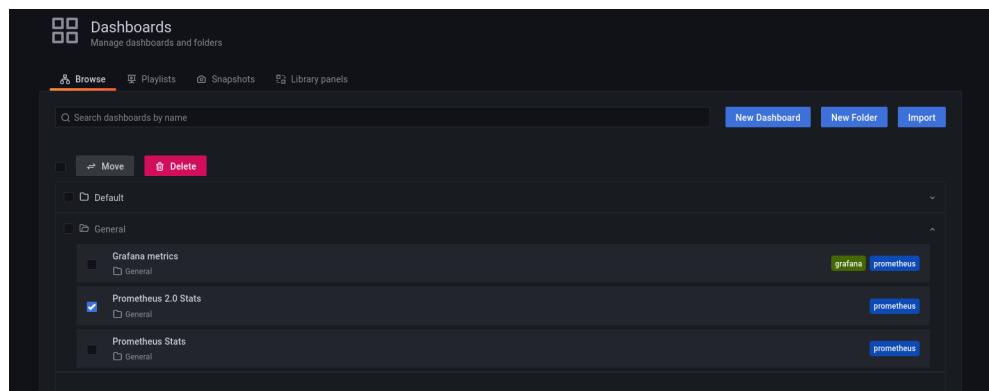


Figura 76: Selección de paneles en Grafana

### 9.3.2. Panel de Kubernetes

Kubernetes dispone de un panel de control propio. Lo instalamos en el clúster siguiendo las indicaciones en la documentación[21]. Este panel de control dispone información sobre cada recurso desplegado en cualquier espacio de nombres.

The screenshot shows the Kubernetes Dashboard interface. On the left, a sidebar lists various resources: Workloads (Deployments), Service, Config and Storage, Cluster. The main area displays 'Workload Status' with three large green circles representing Deployments (Running: 3), Pods (Running: 6), and Replica Sets (Running: 3). Below this, the 'Deployments' section shows a table with three entries:

Name	Images	Labels	Pods	Created
vista-deployment	trastolillo/web	-	3 / 3	2 hours ago
mysql-deployment	trastolillo/mysql	-	1 / 1	10 hours ago
flaskapi-deployment	trastolillo/flask	app: flaskapi	2 / 2	16 hours ago

Below the Deployments table, the 'Pods' section shows a table with one entry:

Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created
vista-deployment-5647c8859b-hdqqj	trastolillo/web	pod-template-hash: 5647c8859b run: vista	worker2	Running	0	-	-	2 hours ago

Figura 77: Kubernetes Dashboard funcionando en el clúster

También dispone de una sección para comprobar el uso de memoria y cpu de cada Pod.

Pods								
Nombre	Imágenes	Etiquetas	Nodo	Estado	Reinicios	Utilización de CPU (núcleos)	Utilización de memoria (octetos)	Fecha de creación
vista-deployment-5647c8859b-9vfhf	trastolillo/web	pod-template-hash: 5647c8859b run: vista	worker2	Running	0	0,00m	9,16Mi	53 minutes ago
vista-deployment-5647c8859b-jmst7	trastolillo/web	pod-template-hash: 5647c8859b run: vista	worker1	Running	0	0,00m	3,60Mi	53 minutes ago
vista-deployment-5647c8859b-z9bwt	trastolillo/web	pod-template-hash: 5647c8859b run: vista	worker3	Running	0	0,00m	3,07Mi	53 minutes ago
flaskapi-deployment-5c5796c755-6ctx2	trastolillo/flask	app: flaskapi pod-template-hash: 5c5796c755	worker3	Running	0	2,00m	14,49Mi	56 minutes ago
flaskapi-deployment-5c5796c755-thz59	trastolillo/flask	app: flaskapi pod-template-hash: 5c5796c755	worker1	Running	0	1,00m	42,49Mi	56 minutes ago
mysql-deployment-675f577d6c-sd2v5	trastolillo/mysql	app: mysql pod-template-hash: 675f577d6c	worker3	Running	0	2,00m	58,63Mi	59 minutes ago

Figura 78: Detalle de los Pods en Kubernetes Dashboard

Es interesante observar cómo los nodos workers consumen más recursos de CPU que los maestros por la gestión de la carga de trabajo de la aplicación. Este detalle es importante tenerlo en cuenta al diseñar los recursos del clúster en un principio.

### 9.3.3. Rancher

K3s, la distribución de Kubernetes empleada en la creación de nuestro clúster, es propiedad de Rancher. En la documentación de Rancher se ofrece un método de instalación bastante detallado[22] a través del gestor de paquetes Helm.

Rancher también dispone de un panel donde podemos observar los recursos consumidos por los distintos objetos del clúster.

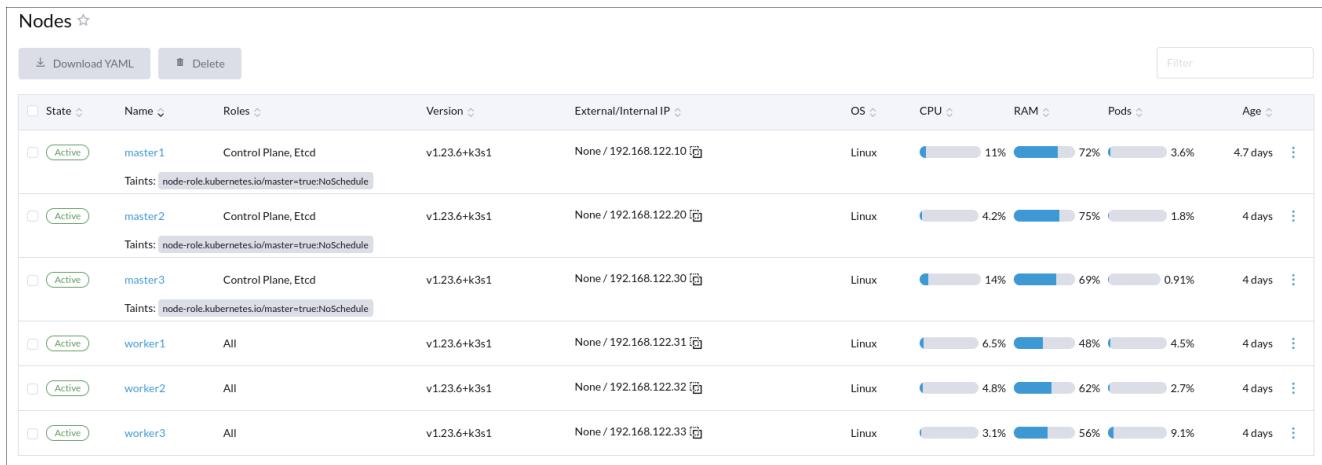


Figura 79: Pantalla de monitorización de Rancher

## ANEXOS

Repository del proyecto: <https://github.com/gabrilov/proyecto>

Repository de Docker Hub: <https://hub.docker.com/u/trastolillo>

### ANEXO I: Aplicación web

#### Vista

##### index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<link rel="stylesheet" href="style.css" />
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.0/jquery.min.js"></script>
<link
rel="stylesheet"
href="https://fonts.sandbox.google.com/css2?
family=Material+Symbols+Outlined:opsz,wght,FILL,GRAD@48,400,0,0"
/>

<script src=".//script.js"></script>
<title>WebApp</title>
</head>

<body>
<header>
<nav>
<ul id="menu">
<li id="modulos">Módulos</li>
<li id="unidades">Unidades</li>
</ul>
</nav>
</header>

<main>
<form id="form-modulo">
<label for="id-modulo">Id</label>
<label id="label2" for="nombre">Nombre</label>
```

```

<label id="label3" for="url">URL</label>
<button class="enviar" type="reset">Limpiar</button>
<input type="text" name="id-modulo" id="id-modulo" required />
<input type="text" name="nombre" id="nombre" required />
<input type="text" name="url" id="url" required />
<button class="enviar" type="submit">Enviar</button>
<h3 class="mensaje">Rellena los campos si quieras que funcione</h3>
</form>

<table>
<thead id="thead"></thead>
<tbody id="tbody"></tbody>
</table>
<h2 class="mensaje-borrado">ELEMENTO BORRADO</h2>

</main>
</body>
</html>

```

**style.css**

```

@import url('https://fonts.googleapis.com/css2?
family=Montserrat:wght@400;500;700&display=swap');

* {
margin: 0;
padding: 0;
}

p {
margin: 0.8em;
text-align: justify;
}

body {
margin: 20px auto;
font-family: 'Montserrat', sans-serif;
background: #ecefef;
max-width: 1200px;
font-size: 1em;
font-weight: 700;
}

header {
grid-column: -1 / 1;
background: #363661;
color: #ecefef;
}

```

```
justify-self: stretch;
text-align: center;
box-shadow: 1px 0 10px #535353;
}

nav ul {
margin: auto;
max-width: 400px;
display: flex;
justify-content: space-between;
align-content: center;
list-style: none;
}

li {
margin: 30px auto;
padding: 10px;
border: 1px solid;
border-radius: 25%;
}

.botones-registro button {
padding: 5px;
}

#modulos,
#tareas,
#unidades {
cursor: pointer;
}

main {
display: grid;
grid-template-columns: repeat(3, 1fr);
gap: 20px;
align-items: start;
}

.boton {
margin: 10px auto;
text-align: center;
}

table {
margin: 20px;
border-collapse: collapse;
grid-column: 1/-1;
}
```

```
table td {
padding: 10px;
}

table td,
table th {
border: 1px solid;
padding: 5px 10px;
}

thead {
background-color: ghostwhite;
}

tr {
background-color: cornsilk;
transition: all 500ms ease;
}

tr:hover {
background-color: darkgrey;
}

.trash {
transition: all 500ms ease;
}

.trash:hover {
background-color: red;
}

#tbody tr {
cursor: pointer;
}

form {
grid-column: 1 / -1;
margin: 20px auto;
text-align: center;
display: none;
}

form div {
margin: 10px auto;
}

form div input {
margin: auto 15px;
}
```

```
#form-modulo {
  display: grid;
  grid-template-columns: 1fr 7fr 1fr 2fr;
  grid-template-rows: 1fr 1fr ;
  justify-items: stretch;
  gap: 10px;
}

label {
  justify-self: start;
  align-self: end;
}

input, textarea {
  font-size: 1.2em;
  word-break: keep-all
}

#id-modulo {
  max-width: 100px;
}

#registrar {
  grid-template-rows: 2 / -1;
  display: none;
}

.enviar {
  font-size: 1.1em;
  /* grid-row: 1 / -1;
  align-self: center;
  justify-self: stretch; */
}

.mensaje {
  grid-column: 1 / -1;
  justify-content: center;
  font-weight: bold;
  color: red;
  display: none;
}

.mensaje-borrado {
  font-weight: bold;
  justify-content: center;
  text-align: center;
  color: red;
  display: none;
```

```

}

.iconos {
  text-align: center;
  cursor: pointer;
}

/* Iconos */
.material-symbols-outlined {
  font-variation-settings:
  'FILL' 0,
  'wght' 400,
  'GRAD' 0,
  'opsz' 48
}

```

**script.js**

```

$(document).ready(function () {
  console.log("Script cargado");
  /***** OBJETOS *****

  const modulo = {
    id_modulo: "",
    modulo: "",
    url: ""
  };

  const unidad = {
    id_modulo: "",
    unidad: "",
    titulo: ""
  };

  /***** VARIABLES *****/
  const URL = "http://192.168.122.231:5000";
  const cabecera = ["modulos", "unidades"];
  const editIconCell =
    '<td class="iconos edit"><span id="editar" class="material-symbols-outlined">edit</span></td>';
  const trashIconCell =
    '<td class="iconos trash"><span class="material-symbols-outlined">delete_forever</span></td>';
  const templateModulos =
    <tr id="${modulo.id}">
      <td id="id-modulo" class="tabla-clickable">${modulo.id}</td>
      <td class="tabla-clickable">${modulo.modulo}</td>
      <td class="tabla-clickable">${modulo.url}</td>
    
```

```

${trashIconCell}
</tr>
` ;
const templateUnidades =
<tr id="${unidad.id_modulo}">
<td id="id-modulo" class="tabla-clickable">${unidad.id_modulo}</td>
<td class="tabla-clickable">${unidad.unidad}</td>
<td class="tabla-clickable">${unidad.titulo}</td>
${trashIconCell}
</tr>
` ;
let tHeadTabla = "<th>Id</th><th>Nombre</th><th>URL</th>";
let isUpdate = false;
let interruptor = "modulos";

***** CARGA INFO INICIO *****

recuperarItems();

***** EVENTOS *****

$("#modulos").click(function (e) {
if (interruptor != "modulos") {
$("#form-modulo")[0].reset();
$("#label2")[0].textContent = "Nombre";
$("#label3")[0].textContent = "URL";
tHeadTabla = "<th>Id</th><th>Nombre</th><th>URL</th>";
interruptor = "modulos";
recuperarItems();
}
});

$("#unidades").click(function (e) {
if (interruptor != "unidades") {
$("#form-modulo")[0].reset();
interruptor = "unidades";
tHeadTabla = "<th>Modulo</th><th>Título</th><th>Unidad</th>";
$("#label2")[0].textContent = "Título";
$("#label3")[0].textContent = "Unidad";
recuperarItems();
}
});

$("button[type='submit']").click(function (e) {
e.preventDefault();
if ($("#id-modulo").val() == "") {
$(".mensaje").css("display", "block");
return;
}
})

```

```

modulo.id_modulo = $("#id-modulo").val();
modulo.modulo = $("#nombre").val();
modulo.url = $("#url").val();
if (interruptor == "unidades") {
    unidad.id_modulo = $("#id-modulo").val();
    unidad.unidad= $("#url").val();
    unidad.titulo = $("#nombre").val();
}
registrarModulo();
$("#form-modulo")[0].reset();
});

$("button[type='reset']").click(function (e) {
    isUpdate = false;
});

/***** FUNCIONES *****/
function recuperarItems() {
    $("#thead").html(tHeadTabla);
    $.ajax({
        type: "GET",
        url: `${URL}/${interruptor}`,
        headers: {
            "access-control-allow-origin": "*",
        },
        dataType: "json",
        success: function (response) {
            let template = '';
            if (interruptor == "modulos") {
                response.modulos.forEach((item) => {
                    template += populateTemplate(item);
                });
            } else {
                response.unidades.forEach((item) => {
                    template += populateTemplate(item);
                });
            }
            $('#tbody').html(template);
        },
        complete: function (data) {
            $(".tabla-clickable").click(function (e) {
                const clave = $(this)[0].parentElement.getAttribute('id');
                isUpdate = true;
                if(interruptor == "unidades") {
                    const claveUnidad = $(this)[0].parentElement.children[1].textContent;
                    llamarUnItem(clave, claveUnidad)
                } else {
                    llamarUnItem(clave);
                }
            });
        }
    });
}

```

```
};

});

$(".trash").click(function (e) {
const clave = $(this)[0].parentElement.getAttribute('id');
if(interruption == "unidades") {
const claveUnidad = $(this)[0].parentElement.children[1].textContent;
borrarModulo(clave, claveUnidad)
} else {
borrarModulo(clave);
}
});
},
});
});

function llamarUnItem(moduloId, unidadId) {
let endpoint = `${URL}/${interruption}/${moduloId}/${unidadId}`;
if (interruption == "modulos") {
endpoint = `${URL}/${interruption}/${moduloId}`;
}
$.ajax({
type: "GET",
url: endpoint,
dataType: "json",
success: function (response) {
if (interruption == "modulos") {
modulo.id_modulo = moduloId;
modulo.modulo = response.modulo.modulo;
modulo.url = response.modulo.url;
} else {
unidad.id_modulo = moduloId;
unidad.unidad = unidadId;
unidad.titulo = response.unidad.titulo;
}
},
complete: function (data) {
if(interruption == "modulos") {
$("#id-modulo").val(modulo.id_modulo);
$("#nombre").val(modulo.modulo);
$("#url").val(modulo.url);
} else {
$("#id-modulo").val(unidad.id_modulo);
$("#nombre").val(unidad.titulo);
$("#url").val(unidad.unidad);
}
},
});
}

function registrarModulo() {
```

```
let item = modulo
let tipo = "POST";
let urlRegistro = `${URL}/modulo`;
let actualizarEndpoint = `${URL}/modulo/${item.id_modulo}`;
if (interruptor == "unidades") {
urlRegistro = `${URL}/unidad`;
item = unidad;
}
if (isUpdate) {
if(interruptor == "unidades") {
actualizarEndpoint = `${URL}/unidad/${item.id_modulo}/${item.unidad}`;
}
urlRegistro = actualizarEndpoint;
tipo = "PUT";
}
$.ajax({
type: tipo,
headers: {
"access-control-allow-origin": "*",
},
url: urlRegistro,
data: JSON.stringify(item),
contentType: "application/json; charset=utf-8",
dataType: "json",
success: function (response) {
recuperarItems();
},
});
isUpdate = false;
}

// TODO: De aquí en adelante

function borrarModulo(modId, unidadId) {
let endpoint = `${URL}/modulo/${modId}`;
if(interruptor == "unidades") {
endpoint = `${URL}/unidad/${modId}/${unidadId}`
}
$.ajax({
type: 'DELETE',
url: endpoint,
dataType: "json",
success: function (response) {
if (response.mensaje != "") {
recuperarItems();
}
},
});
}
```

```
function populateTemplate(item) {
  if (interruptor == "modulos") {
    return `
<tr id="${item.id}>
<td id="id-modulo" class="tabla-clickable">${item.id}</td>
<td class="tabla-clickable">${item.modulo}</td>
<td class="tabla-clickable">${item.url}</td>
${trashIconCell}
</tr>
`;
  } else if (interruptor == "unidades") {
    return `
<tr id="${item.id_modulo}>
<td id="id-modulo" class="tabla-clickable">${item.id_modulo}</td>
<td class="tabla-clickable">${item.unidad}</td>
<td class="tabla-clickable">${item.titulo}</td>
${trashIconCell}
</tr>
`;
  }
}
});
```

## API

### app.py

```

import os
from flask import Flask
from flask_mysqldb import MySQL
from flask_cors import CORS

app=Flask(__name__)
# app.config['MYSQL_HOST'] = "mysql"
app.config['MYSQL_HOST'] = os.getenv('host-mysql')
# app.config['MYSQL_PASSWORD'] = "root"
app.config['MYSQL_PASSWORD'] = os.getenv('password-mysql')
app.config['MYSQL_DB'] = "modulos"

conexion=MySQL(app)

CORS(app)

def pagina_no_encontrada(error):
    return "error", 404

if __name__ == '__main__':
# app.config.from_object(config['development'])
    app.register_error_handler(404, pagina_no_encontrada)
    app.run()

from unidades import un
from modulos import mod
app.register_blueprint(mod)
app.register_blueprint(un)

```

### modulos.py

```

from flask import Blueprint, jsonify, request

mod = Blueprint('modulos', __name__)
from app import conexion

@mod.get('/modulos')
def listar_modulos():
    try:
        cursor = conexion.connection.cursor()
        sql = 'SELECT * FROM modulos'
        cursor.execute(sql)
        datos = cursor.fetchall()
        modulos = []

```

```

for fila in datos:
    modulo = {'id':fila[0], 'modulo':fila[1], 'url':fila[2]}
    modulos.append(modulo)
return jsonify({'modulos':modulos, 'mensaje': 'Modulos listados'})
except Exception as ex:
    print('_____')
    print(ex)
    print('_____')
    return jsonify({'mensaje': 'error', 'error': str(ex)})

@mod.get('/modulos/<id_modulo>')
def modulo(id_modulo):
    try:
        cursor = conexion.connection.cursor()
        sql = "SELECT * FROM modulos WHERE id_modulo= '{0}'".format(id_modulo)
        cursor.execute(sql)
        datos = cursor.fetchone()
        if datos != None:
            modulo = {'id':datos[0], 'modulo':datos[1], 'url':datos[2]}
            return jsonify({'modulo':modulo, 'mensaje': 'Modulo encontrado'})
        else:
            return jsonify({'mensaje': 'Este módulo no existe'})
    except Exception as ex:
        print('_____')
        print(ex)
        print('_____')
        return jsonify({'mensaje': 'error', 'error': str(ex)})

@mod.post('/modulo')
def registrar_modulo():
    try:
        cursor = conexion.connection.cursor()
        sql = """INSERT INTO modulos (id_modulo, modulo, url)
VALUES ('{0}', '{1}', '{2}')""".format(request.json['id_modulo'],
request.json['modulo'], request.json['url'])
        cursor.execute(sql)
        conexion.connection.commit()
        return jsonify({'mensaje': 'Registrado'})
    except Exception as ex:
        print('_____')
        print(ex)
        print('_____')
        return jsonify({'mensaje': 'error', 'error': str(ex)})

@mod.delete('/modulo/<id_modulo>')
def eliminar_modulo(id_modulo):
    try:
        cursor = conexion.connection.cursor()
        sql = "DELETE FROM modulos WHERE id_modulo= '{0}'".format(id_modulo)
    
```

```

cursor.execute(sql)
conexion.connection.commit()
return jsonify({'mensaje': 'Registro borrado'})
except Exception as ex:
print('_____')
print(ex)
print('_____')
return jsonify({'mensaje': 'error', 'error': str(ex)})

@mod.put('/modulo/<id_modulo>')
def actualizar_modulo(id_modulo):
print(conexion)
try:
cursor = conexion.connection.cursor()
sql = 'UPDATE modulos SET modulo="{0}", url="{1}" WHERE id_modulo="{2}"'.format(
request.json['modulo'],
request.json['url'],
id_modulo
)
print(sql)
cursor.execute(sql)
conexion.connection.commit()
return jsonify({'Resultado': 'Actualizado'})
except Exception as ex:
print('_____')
print(ex)
print('_____')
return jsonify({'mensaje': 'error', 'error': str(ex)})

```

**unidades.py**

```

from flask import Blueprint, jsonify, request

un = Blueprint('unidades', __name__)
from app import conexion

@un.get('/unidades')
def listar_unidades():
try:
cursor = conexion.connection.cursor()
sql = 'SELECT * FROM unidades'
cursor.execute(sql)
datos = cursor.fetchall()
unidades = []
for fila in datos:
unidad = {'id_modulo':fila[0], 'unidad':fila[1], 'titulo':fila[2]}
unidades.append(unidad)
return jsonify({'unidades':unidades, 'mensaje': 'Unidades listadas'})

```

```

except Exception as ex:
    print('-----')
    print(ex)
    print('-----')
    return jsonify({'mensaje': 'error', 'error': str(ex)})

@un.get('/unidades/<id_modulo>/<unidad>')
def modulo(id_modulo, unidad):
    try:
        cursor = conexion.connection.cursor()
        sql = "SELECT * FROM unidades WHERE id_modulo='{0}' and \
        unidad={1}".format(id_modulo, unidad)
        cursor.execute(sql)
        datos = cursor.fetchone()
        if datos != None:
            result = {'id_modulo':datos[0], 'unidad':datos[1], 'titulo':datos[2]}
            return jsonify({'unidad':result, 'mensaje': 'Unidad encontrada'})
        else:
            return jsonify({'mensaje': 'Este módulo no existe'})
    except Exception as ex:
        print('-----')
        print(ex)
        print('-----')
        return jsonify({'mensaje': 'error', 'error': str(ex)})


@un.post('/unidad')
def registrar_modulo():
    try:
        cursor = conexion.connection.cursor()
        sql = """INSERT INTO unidades (id_modulo, unidad, titulo) \
        VALUES ('{0}', '{1}', \
        '{2}')""".format(request.json['id_modulo'],request.json['unidad'],\
        request.json['titulo'])
        cursor.execute(sql)
        conexion.connection.commit()
        return jsonify({'mensaje': 'Registrado'})
    except Exception as ex:
        print('-----')
        print(ex)
        print('-----')
        return jsonify({'mensaje': 'error', 'error': str(ex)})


@un.delete('/unidad/<id_modulo>/<unidad>')
def eliminar_modulo(id_modulo, unidad):
    try:
        cursor = conexion.connection.cursor()
        sql = "DELETE FROM unidades WHERE id_modulo='{0}' and \
        unidad={1}".format(id_modulo, unidad)
        cursor.execute(sql)

```

```

conexion.connection.commit()
return jsonify({'mensaje': 'Registro borrado'})
except Exception as ex:
print('_____')
print(ex)
print('_____')
return jsonify({'mensaje': 'error', 'error': str(ex)})

@un.put('/unidad/<id_modulo>/<unidad>')
def actualizar_modulo(id_modulo, unidad):
try:
cursor = conexion.connection.cursor()
sql = 'UPDATE unidades SET titulo={0} WHERE id_modulo={1} and
unidad={2}'.format(
request.json['titulo'],
id_modulo,
unidad
)
print(sql)
cursor.execute(sql)
conexion.connection.commit()
return jsonify({'Resultado': 'Actualizado'})
except Exception as ex:
print('_____')
print(ex)
print('_____')
return jsonify({'mensaje': 'error', 'error': str(ex)})

```

**requirements.txt**

```

click==8.1.2
Flask==2.1.1
Flask-Cors==3.0.10
Flask-MYSQLdb==1.0.1
itsdangerous==2.1.2
Jinja2==3.1.1
MarkupSafe==2.1.1
mysqlclient==2.1.0
six==1.16.0
Werkzeug==2.1.1

```

## Base de datos

### schema.sql

```
create database if not exists modulos;

use modulos;

create table if not exists modulos (
    id_modulo varchar(6) primary key not null,
    modulo varchar(200) not null,
    url varchar(250)
) engine = InnoDB;

create table if not exists unidades (
    id_modulo varchar(6) not null,
    unidad tinyint not null,
    titulo varchar(255) not null,
    primary key (id_modulo, unidad),
    constraint fk_unidades_modulos foreign key(id_modulo) references
modulos(id_modulo) on delete restrict on update cascade
) engine = InnoDB;

/* **** Dummy data **** */
insert into
modulos
values(
"ASO",
"Administración de Sistemas Operativos",
"#"
),
(
"ASGBD",
"Administración de Sistemas Gestores de Base de Datos",
"#"
),
(
"SRI",
"Servicios de Red e Internet",
"#"
),
(
"IAW",
"Implantación de Aplicaciones Web",
"#"
),
(
"SAD",
```

```
"Seguridad y Alta Disponibilidad",
"#"
),
(
"EIE",
"Empresa e Iniciativa Emprendedora",
"#"
);
insert into
unidades
values
(
"ASO",
1,
"Servicios de acceso y administración remota"
),
(
"ASO",
2,
"Administración de servidores de impresión"
),
(
"ASO",
3,
"Administración de servicios de directorio"
),
(
"IAW",
1,
"Instalación de servidores de aplicaciones web"
),
(
"IAW",
2,
"Programación de documentos web utilizando lenguajes de script de servidor"
),
(
"IAW",
3,
"Acceso a bases de datos desde lenguajes de script de servidor"
);
```

## ANEXO II: Docker

### Dockerfile.mysql

```
FROM mysql:8.0
ENV LANG=C.UTF_8
COPY ./schema.sql /docker-entrypoint-initdb.d/:ro
```

### Dockerfile.flask

```
FROM python:bullseye
WORKDIR /app
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV FLASK_ENV=development
ENV DEBIAN_FRONTEND=noninteractive
RUN apt update && apt install -y python3-mysqldb && rm -rf /var/lib/apt/lists/*
COPY ./backend/ .
RUN pip3 install -r requirements.txt
CMD ["flask", "run"]
```

### Dockefile.nginx

```
FROM nginx:1.21
WORKDIR /usr/share/nginx/html
COPY ./html/ .
```

## ANEXO III: Utilidades

### .ssh/config

```
Host m1
    Hostname 192.168.122.10
    User root

Host m2
    Hostname 192.168.122.20
    User root

Host m3
    Hostname 192.168.122.30
    User root

Host w1
    Hostname 192.168.122.31
    User root

Host w2
    Hostname 192.168.122.32
    User root

Host w3
    Hostname 192.168.122.33
    User root
```

### nodes.sh

```
#!/bin/bash

# Envía un comando a todos los nodos
for i in {1..3}
do
echo
echo "_____"
echo "Enviado a m$i: $1"
echo "_____"
ssh m$i $1
echo
echo "_____"
echo "Enviado a w$i: $1"
echo "_____"
ssh w$i $1
done
```

## ANEXO IV: Manifiestos

### mysql-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: mysql-secret
  type: Opaque
data:
  password: cm9vdA==
```

### mysql-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-configmap
data:
  database_url: mysql-service
```

### mysql-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - image: trastolillo/mysql
```

```
name: mysql
env:
- name: MYSQL_ROOT_PASSWORD
  valueFrom:
    secretKeyRef:
      name: mysql-secret
      key: password
  ports:
- containerPort: 3306
  name: mysql
  volumeMounts:
- name: mysql-persistent-storage
  mountPath: /var/lib/mysql
  volumes:
- name: mysql-persistent-storage
  persistentVolumeClaim:
    claimName: mysql-pvc

  apiVersion: v1
  kind: Service
  metadata:
    name: mysql-service
  spec:
    ports:
- protocol: TCP
  port: 3306
  targetPort: 3306
  selector:
    app: mysql
```

### flask-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flaskapi-deployment
  labels:
    app: flaskapi
spec:
  replicas: 2
  selector:
    matchLabels:
      app: flaskapi
  template:
    metadata:
      labels:
```

```
app: flaskapi
spec:
  containers:
    - name: flaskapi
      image: trastolillo/flask
      imagePullPolicy: IfNotPresent
      ports:
        - containerPort: 5000
      env:
        - name: password-mysql
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: password
        - name: host-mysql
          valueFrom:
            configMapKeyRef:
              name: mysql-configmap
              key: database_url

  -
  apiVersion: v1
  kind: Service
  metadata:
    name: flask-service
  spec:
    ports:
      - port: 5000
        protocol: TCP
        targetPort: 5000
        nodePort: 31000
    selector:
      app: flaskapi
    type: LoadBalancer
```

### web-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vista-deployment
spec:
  selector:
    matchLabels:
      run: vista
  replicas: 3
```

```
template:
metadata:
labels:
run: vista
spec:
containers:
- name: vista
image: trastolillo/web
imagePullPolicy: IfNotPresent
ports:
- containerPort: 80
resources:
requests:
memory: "300Mi"
cpu: "500m"
limits:
memory: "500Mi"
cpu: "700m"

apiVersion: v1
kind: Service
metadata:
name: vista-svc
labels:
run: vista
spec:
ports:
- port: 80
targetPort: 80
protocol: TCP
selector:
run: vista
```

### web-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: web-ingress
spec:
rules:
- host: web.home
http:
paths:
```

```
- path: /
pathType: Prefix
backend:
service:
name: vista-svc
port:
number: 80
ingressClassName: nginx
```

## Referencias

- [1] Israel, «Arquitectura de microservicios vs arquitectura monolítica», *Viewnext*, 3 de mayo de 2018. <https://www.viewnext.com/arquitectura-de-microservicios-vs-arquitectura-monolitica/> (accedido 6 de abril de 2022). ← Volver
- [2] «Kubernetes». <https://kubernetes.io/es/> (accedido 3 de abril de 2022). ← Volver
- [3] «Servicio de Kubernetes administrado – Amazon EKS – Amazon Web Services», *Amazon Web Services, Inc.* <https://aws.amazon.com/es/eks/> (accedido 3 de abril de 2022). ← Volver
- [4] «Kubernetes - Google Kubernetes Engine (GKE)», *Google Cloud*. <https://cloud.google.com/kubernetes-engine> (accedido 3 de abril de 2022). ← Volver
- [5] «CI/CD», *Wikipedia, la enciclopedia libre*. 18 de enero de 2022. Accedido: 3 de abril de 2022. [En línea]. Disponible en: <https://es.wikipedia.org/w/index.php?title=CI/CD&oldid=141050014> ← Volver
- [6] «Welcome! - Minikube», *minikube*. <https://minikube.sigs.k8s.io/docs/> (accedido 5 de abril de 2022). ← Volver
- [7] «Innovate Everywhere», *Rancher Labs*. <http://rancher.com> (accedido 6 de abril de 2022). ← Volver
- [8] «Installation Requirements», *Rancher Labs*. <https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/> (accedido 4 de mayo de 2022). ← Volver
- [9] *CNI - the Container Network Interface*. CNI, 2022. Accedido: 7 de mayo de 2022. [En línea]. Disponible en: <https://github.com/containernetworking/cni> ← Volver
- [10] *flannel*. flannel-io, 2022. Accedido: 7 de mayo de 2022. [En línea]. Disponible en: <https://github.com/flannel-io/flannel> ← Volver
- [11] «Migra a microservicios desde una aplicación monolítica | Entorno estándar de App Engine para Python 2», *Google Cloud*. <https://cloud.google.com/appengine/docs/standard/python/microservice-migration?hl=es-419> (accedido 7 de abril de 2022). ← Volver
- [12] A. Ellis, *k3sup (said 'ketchup')*. 2022. Accedido: 23 de mayo de 2022. [En línea]. Disponible en: <https://github.com/alexellis/k3sup> ← Volver
- [13] «MetalLB, bare metal load-balancer for Kubernetes». <https://metallb.org/> (accedido 2 de junio de 2022). ← Volver
- [14] «Longhorn | What is Longhorn?», *Longhorn*. <https://longhorn.io/docs/1.2.4/what-is-longhorn/> (accedido 2 de junio de 2022). ← Volver
- [15] «Run a Replicated Stateful Application», *Kubernetes*. <https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/> (accedido 5 de junio de 2022). ← Volver
- [16] «To run or not to run a database on Kubernetes: What to consider», *Google Cloud Blog*. <https://cloud.google.com/blog/products/databases/to-run-or-not-to-run-a-database-on-kubernetes-what-to-consider/> (accedido 5 de junio de 2022). ← Volver
- [17] «Installation Guide - NGINX Ingress Controller». <https://kubernetes.github.io/ingress-nginx/deploy/> (accedido 5 de junio de 2022). ← Volver
- [18] Prometheus, «Prometheus - Monitoring system & time series database». <https://prometheus.io/> (accedido 5 de junio de 2022). ← Volver
- [19] «Grafana: The open observability platform», *Grafana Labs*. <https://grafana.com/> (accedido 5 de junio de 2022). ← Volver

- [20]«kube-prometheus/manifests at main · prometheus-operator/kube-prometheus», *GitHub*.  
<https://github.com/prometheus-operator/kube-prometheus> (accedido 5 de junio de 2022). ← Volver
- [21]«Kubernetes Dashboard», *Rancher Labs*. <https://rancher.com/docs/k3s/latest/en/installation/kube-dashboard/> (accedido 5 de junio de 2022). ← Volver
- [22]«Install/Upgrade Rancher on a Kubernetes Cluster», *Rancher Labs*. ← Volver  
<https://rancher.com/docs/rancher/v2.5/en/installation/install-rancher-on-k8s/> (accedido 5 de junio de 2022). ← Volver