Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

ADMINISTRACIÓN DE SISTEMAS
4º curso, grupo 46
Primer cuatrimestre

APLICACIÓN MENDITRACK

Sergio Martín

Bilbao, 3 de diciembre de 2023

Índice

Índice de figuras

1. Introducción

1.1. Definición de la tarea

En este proyecto se propone construir una aplicación web funcional compuesta, al menos, por un servidor web, un servidor de bases de datos y un servicio adicional de libre elección. La aplicación debe de estar compuesta, al menos, por estos tres servicios y debe de ser desplegable mediante un fichero *"docker-compose.yml"*. Finalmente, para los datos generados por la aplicación y almacenados en la base de datos se debe gestionar la persistencia, almacenando los datos en una ubicación ajena al contenedor para permitir las copias de seguridad.

1.2. Tareas realizadas

- Desarrollar una aplicación funcional ✓
- Utilizar al menos 3 imágenes como base ✓
- Desplegar la aplicación en un entorno *Docker* ✓
- Desplegar la aplicación en un entorno *k8s* ✓
- Incluir contenedores/imágenes extra ✓
- Utilizar funcionalidades de *Docker* y *k8s* no vistas en clase
  - Función *healthcheck* de *Docker* ✓
  - Objeto *ConfigMap* de *k8s* ✓

1.3. Presentación de la idea

El sistema web a desarrollar, que toma el nombre de ***MendiTrack***, funcionará como un portal de rutas de montaña, donde los usuarios podrán añadir sus trayectos para tenerlos registrados. De igual forma, con la intención de crear una comunidad, aquellos usuarios que así lo deseen podrán compartir algunas de sus rutas con el resto de participantes en la aplicación. (Ver figura 1)

Por cada ruta se mostrarán varias características tales como el grado de dificultad, la longitud y el desnivel de la misma con la intención de que con un vistazo rápido se pueda evaluar la ruta que otro usuario ha realizado.

La aplicación no registrará datos complejos como archivos *".gpx"* que contengan el *track* de ruta. No obstante, el usuario podrá adjuntar un enlace a algún portal (*Strava, Wikiloc, etc.*) donde esos datos estén disponibles.

Para hacer uso de la aplicación es mandatorio estar registrado e iniciar sesión. Además, para mayor seguridad, tras completar el proceso de registro, el usuario debe iniciar sesión manualmente para acceder a la aplicación.

Una vez el usuario se ha registrado se le muestra la página principal, donde están visibles todas las rutas públicas registradas en el servicio web. Las rutas se muestran consecutivamente con sus respectivas especificaciones, y con un botón de acceso a información complementaria en alguna aplicación externa si es que el propietario de la ruta ha querido añadirlo.

Desde la página principal, haciendo uso de la barra de tareas en la parte superior de la pantalla, el usuario puede desplazarse a su perfil privado. La vista que se abre a continuación es exclusiva y privada para cada usuario, en ella aparecen sus datos personales y todas las rutas que el propio usuario haya añadido a ***MendiTrack***. Para cada una de estas rutas, el usuario desde su perfil privado puede realizar dos acciones exclusivas del propietario de la ruta: editarla o eliminarla. Además, es en esta vista donde aparece la opción de añadir nuevas rutas.

Finalmente, por motivos de seguridad, si el usuario excede un tiempo de 3 minutos de inactividad, su sesión se cierra automáticamente. Esto provoca que ante cualquier interacción se le solicite al usuario volver a iniciar sesión.

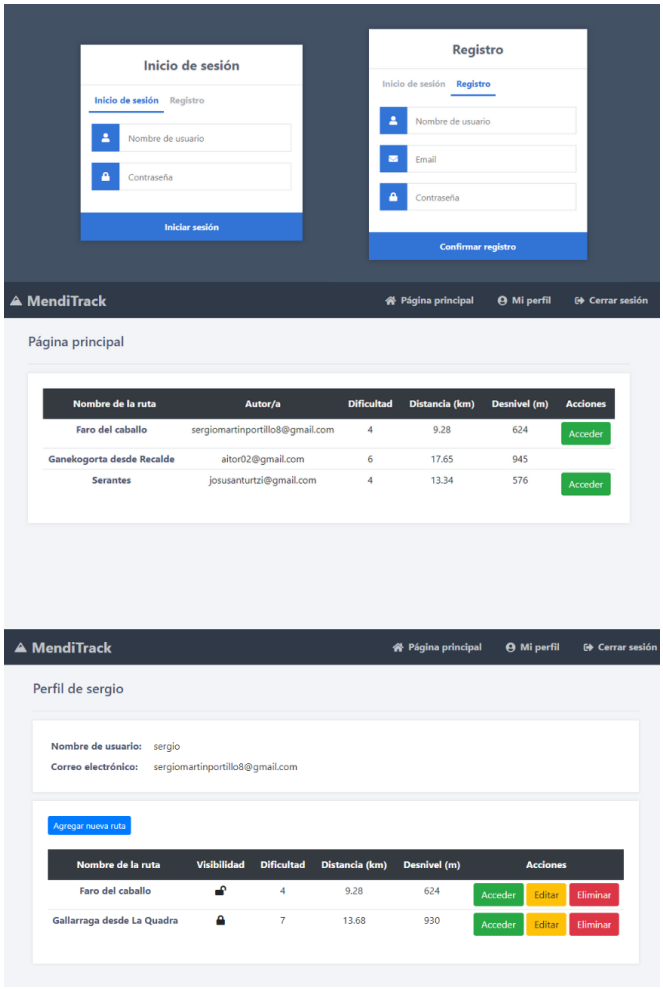

Figura 1: Interfaz visual de la aplicación web

2. Repositorio de *GitHub*

Enlace al repositorio de *GitHub* con el código desarrollado en esta tarea:

https://github.com/Sergiom8m/Sistema_Web_Docker

3. Especificaciones técnicas y arquitectura interna

La aplicación es completamente funcional gracias a un total de 6 servicios diferentes (ver figura 2), siendo algunos de ellos servicios principales y otros auxiliares. El tronco de la aplicación está formado por 3 contenedores que contienen imágenes base de *Python* y *MySQL*. Mientras tanto, los otros tres contenedores incluidos en el entorno *Docker* se pueden considerar microservicios que añaden pequeñas funcionalidades a la web con el fin de que la experiencia del usuario sea más agradable.

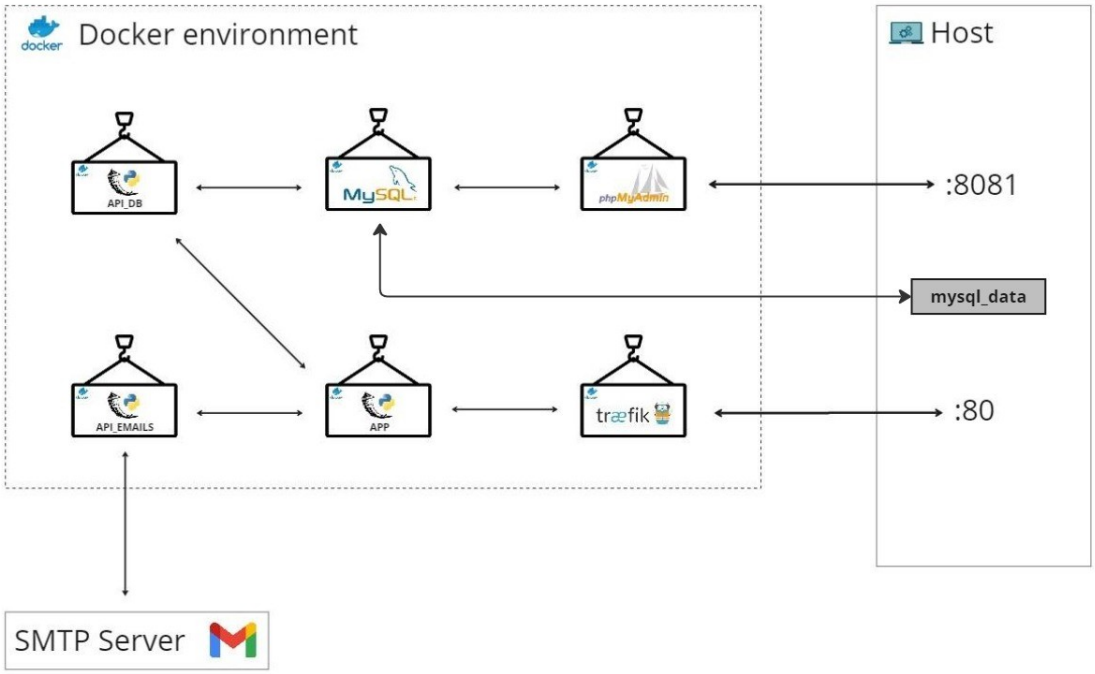

Figura 2: Arquitectura de la aplicación web

3.0.1. Servicio *app*

Este servicio ejecuta sobre una base de *Python* una aplicación del *framework Flask* [3]. Está destinado a gestionar la interacción de los usuarios con la aplicación a través de una interfaz web. Funciona como el componente principal del sistema, manejando el registro de usuarios, autenticación, gestión de perfiles, rutas y la comunicación con otros servicios del entorno.

En resumen, el servicio *app* sirve la aplicación web a los usuarios y responde a la interacción de los usuarios con la interfaz gráfica, valiéndose de los demás servicios para ello.

3.0.2. Servicio *db*

Como su propio nombre indica, en este contenedor se ejecuta un servidor de bases de datos, *MySQL* [5] en este caso concreto. Actúa como una parte esencial del sistema, proporcionando la capacidad de almacenamiento y recuperación de información necesaria para el funcionamiento correcto de la aplicación.

Adicionalmente, para gestionar la persistencia de los datos, la información contenida en este contenedor ha sido abstraída en un volumen externo. De esta forma, los datos contenidos en la base de datos son alcanzables desde ubicaciones ajenas al contenedor.

3.0.3. Servicio *api_db*

Este servicio se encarga de proporcionar una interfaz entre la aplicación web (servicio *app*) y la base de datos (servicio *db*). Esencialmente, maneja las operaciones de lectura y escritura en la base de datos en respuesta a las solicitudes provenientes de la aplicación. Proporciona una capa de abstracción que facilita el acceso a los datos almacenados en el contenedor de la base de datos.

Por otra parte, se considera una gran aportación en cuestiones de escalabilidad y seguridad. En primer lugar, supone que el manejo de la base de datos y el proceso de *hasheo* de las contraseñas quede alejado de la aplicación principal, facilitando así la opción de cambiar el servidor de base de datos o el algoritmo criptográfico sin necesidad de modificar la aplicación.

Además, haciéndolo de esta forma el servicio *app* que queda expuesto al usuario no tiene por qué tener acceso a la base de datos, únicamente hará peticiones a este servicio para que se hagan transacciones en la base de datos.

3.0.4. Servicio *api_emails*

Este microservicio se dedica al envío de correos electrónicos de confirmación después de que un usuario se registra exitosamente en la aplicación. Actúa como un componente independiente para manejar las comunicaciones por correo electrónico, proporcionando una funcionalidad específica y separada de la lógica principal de la aplicación.

Nuevamente, será un servicio aislado para el usuario, es el servicio *app* el encargado de realizar peticiones sobre esta interfaz de mensajería. Esta interfaz de *emails* se conecta mediante una librería de *Python* [7] a un servidor *SMTP*, el de *Gmail*, en este caso, para enviar los mensajes.

3.0.5. Servicio *Traefik*

Este servicio ejecuta *Traefik* [8], actuando como un enrutador y proporcionando funcionalidades de *proxy inverso*. Facilita la gestión de las solicitudes de red, direccionando el tráfico hacia los distintos servicios basándose en reglas de enrutamiento configuradas. Para este caso concreto, este

servicio es útil para servir la aplicación en el puerto 80, cuando por defecto se ejecuta en el puerto 5000. (Ver figura 3)

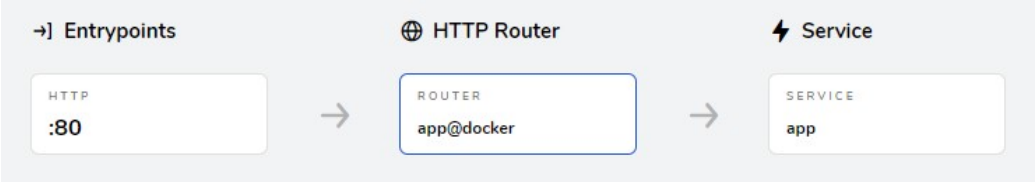

Figura 3: Diagrama de enrutamiento

Además de eso, en cuestiones de escalabilidad puede ser de gran ayuda, puesto que si se decide duplicar el número de instancias del servicio *app* con la intención de soportar un mayor tráfico de usuarios, *Traefik* sin necesidad de configuraciones densas actuaria como *load balancer* distribuyendo el tráfico entre las instancias de *app* funcionales. (Ver figura 4) [9]

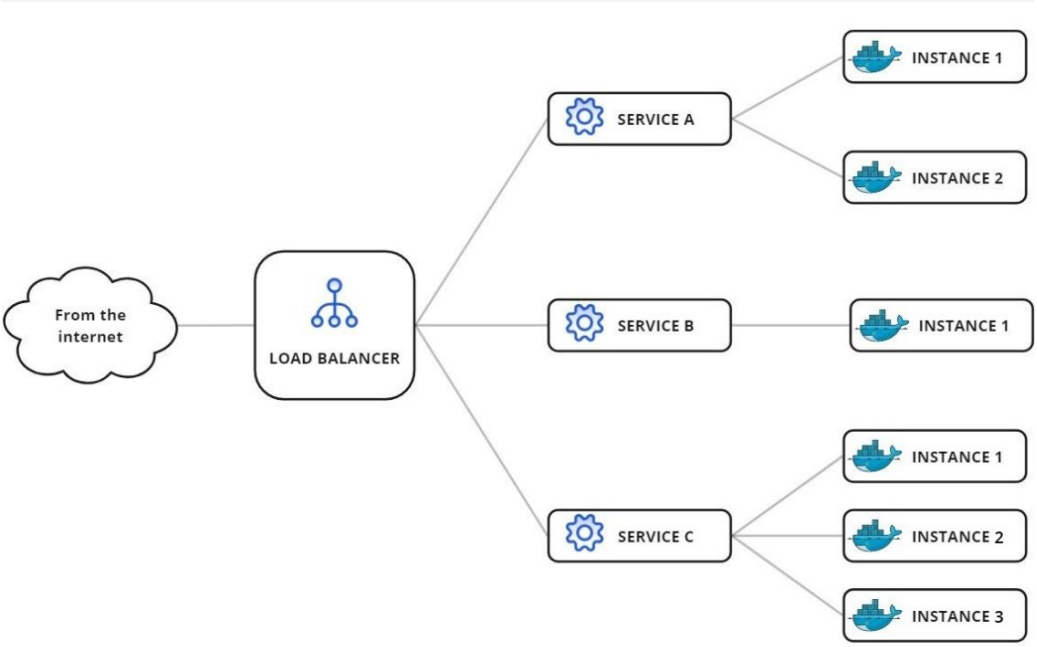

Figura 4: Diagrama *load balancing*

3.0.6. Servicio *PHPMyAdmin*

Este servicio implementa *PHPMyAdmin* [6], una interfaz gráfica de usuario para administrar y manejar visualmente la base de datos *MySQL*. Proporciona una manera conveniente de interactuar con la base de datos a través de una interfaz web, permitiendo realizar tareas de administración de datos sin la necesidad de comandos *SQL* directos.

De cara al usuario no es un servicio útil, pues no conviene que los usuarios tengan acceso a la base de datos. Sin embargo, desde el perfil del desarrollador puede ser de gran ayuda poder gestionar

la base de datos mediante una interfaz gráfica.

4. Funcionalidad *healthcheck*

En la figura 2 quedan expuestas tanto las relaciones internas entre servicios como las relaciones de algunos servicios con el exterior. Cuando la aplicación se despliega desde cero, esas relaciones se convierten en dependencias (ver figura 5), y son algo a respetar y tener él cuenta. De lo contrario, ocurrirán casos donde las peticiones entre servicios no funcionen debido a que uno de los servicios no se haya puesto en funcionamiento aún.

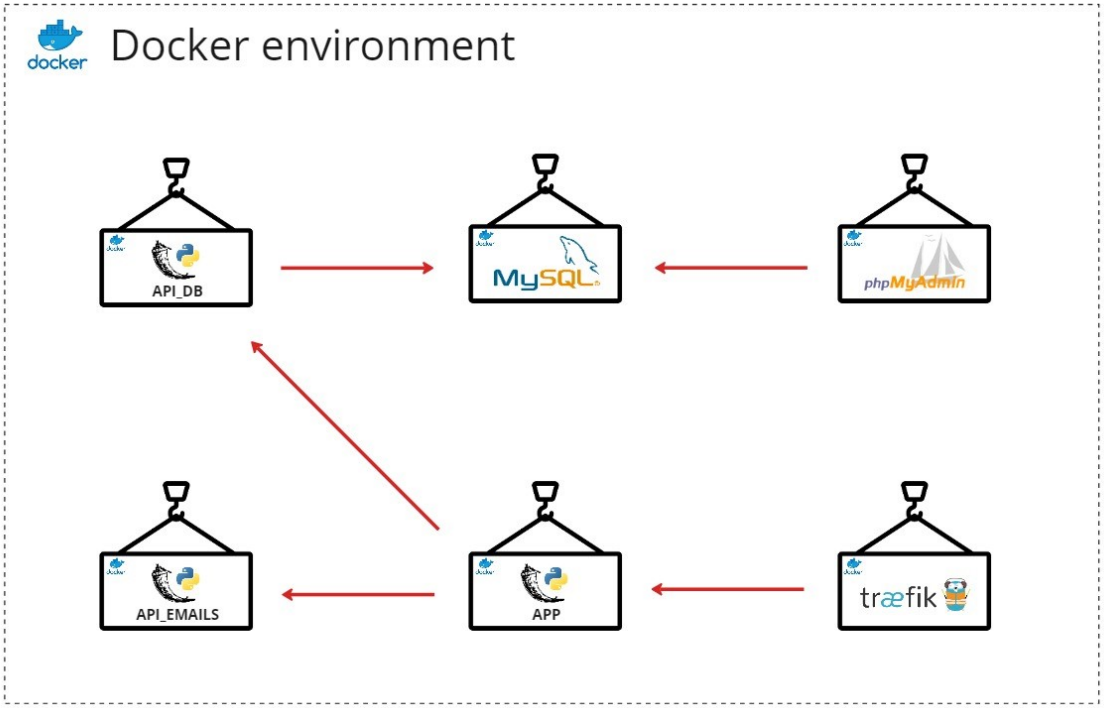

Figura 5: Dependencias entre servicios dentro del entorno *Docker Compose*

Por otra parte, que un contenedor esté en marcha no necesariamente implica que el servicio que ese contenedor ofrece esté disponible. Un ejemplo muy claro de eso en este caso es el servicio *db*. El contenedor que despliega ese servicio se pone en marcha instantáneamente, no obstante, hasta que el servicio *MySQL* es funcional, hay un *delay* de unos segundos.

Para evitar este tipo de conflictos se establecen dependencias en el archivo de configuración de *Docker Compose*. Lo más común es emplear la clave *depends on*, sin embargo, esta da lugar al problema que se acaba de mencionar. Para evitar que un contenedor se considere activo cuando su servicio aún no lo está, se puede ejecutar un *healthcheck* [2] sobre el contenedor.

Aplicar un *healthcheck* supone ejecutar un comando dentro del contenedor a valorar y ver si

ese comando tiene la respuesta esperada o no. Si ocurre algún error o la respuesta no es válida, el *healthcheck* se dará por negativo. De modo que los contenedores que requieran que el servicio actual tenga la clave *healthy* no arrancaran hasta que el *healthcheck* tenga un resultado positivo. Si eso no ocurre, la aplicación no se desplegará.

5. Despliegue con *Kubernetes*

Lo que se ha expuesto hasta el momento requiere de los servicios de *Docker Compose* para desplegar la aplicación. Sin embargo, existen diferentes servicios de orquestación de contenedores, uno de los más usados es *Kubernetes k8s*. Estas dos herramientas proporcionan características diferentes, pero la esencia viene a ser la misma. En el caso de *k8s* destacan sus opciones de escalado y gestión de recursos.

Para desplegar la aplicación ***MendiTrack*** mediante este servicio no se ha de implementar nada nuevo. Simplemente, se precisa de una nueva configuración de despliegue. (Ver figura 6)

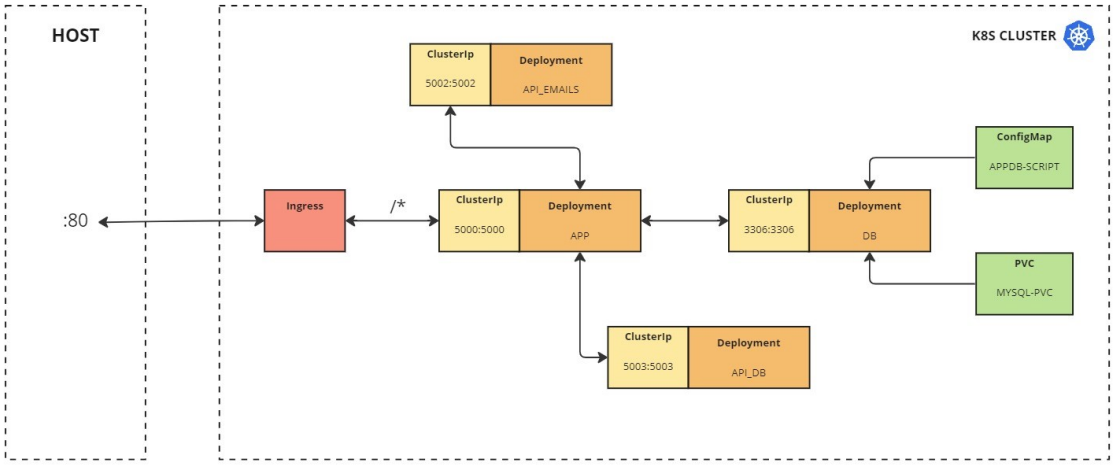

Figura 6: Arquitectura interna del clúster *Kubernetes*

Como se ha mencionado, los servicios que contiene la aplicación siguen siendo los mismos. No obstante, en esta ocasión se han utilizado diferentes objetos del servicio *kubernetes* para su despliegue.

Por una parte, todos los servicios (contenedores) de la aplicación web se despliegan mediante objetos *deployment*. Este objeto sirve para mantener un conjunto de *pods* idénticos, asegurando que todos tienen la configuración correcta y necesaria y todos están en marcha. De modo que si en algún momento uno de los *pod* deja de funcionar, el objeto *deployment* se encarga de poner en marcha otro que lo sustituya.

A su vez, cada objeto *deployment* tiene asociado un objeto *ClusterIP* que sirve principalmente para exponer el contenido y los servicios de ese contenedor al resto de los contendores en el entorno *k8s*.

Hasta el momento, con los objetos mencionados no es posible exponer servicios al exterior del clúster *k8s*. La forma más eficiente de hacerlo es empleando el objeto *ingress*. Este objeto redirige el tráfico desde el exterior del clúster al interior en función de la ruta que el usuario haya solicitado en el navegador. En el caso de esta aplicación solo se ha expuesto un servicio al exterior, sin embargo, si hubiese más servicios, un único *ingress* podría hacerse cargo de redireccionar el tráfico a los diferentes servicios.

Por último, para la configuración del servicio *MySQL* se requiere de dos elementos extra. En primer lugar, con el objetivo de crear una base de datos y su esqueleto en el servicio de base de datos desplegado, se ha configurado un objeto *ConfigMap*. Este elemento únicamente contiene una cadena de texto con las instrucciones en lenguaje *SQL* que se ejecutan cuando el *ConfigMap* es cargado en el contendor de *MySQL*.

Finalmente, se ha empleado un *Persistent Volume Claim (PVC)* para gestionar la persistencia de datos en la aplicación. En algunos motores de *k8s* la creación de volúmenes persistentes se debe hacer manualmente, en el caso de *Kubernetes Engine* de Google la creación es automática con el uso de *PVC*s. La implementación de la persistencia contribuye a que si el *pod* de la base de datos colapsa y se reinicia, los datos almacenados en ese contendor no se pierdan.

6. Declaración sobre el uso de asistentes basados en IA

En el desarrollo de esta tarea se han empleado diferentes asistentes basados en inteligencia artificial. Ejemplo de ello son *ChatGPT* [1] o *Bard* [4], sin embargo, se ha tratado de utilizarlos como herramientas complementarias para aclarar dudas o encontrar información concreta de manera rápida. Se ha evitado utilizar en estos asistentes *prompts* para solucionar problemas o tareas complejas.

Referencias

[1] *ChatGPT*. URL: `https://chat.openai.com/`.

[2] *Docker Engine official docs - Healthcheck feature*. URL: `https://docs.docker.com/engine/reference/builder/#healthcheck`.

[3] *Flask user guide*. URL: `https://flask.palletsprojects.com/en/3.0.x/#user-s-guide`.

[4] *Google Bard*. URL: `https://bard.google.com/chat?hl=es`.

[5] *MySQL image documentation in DockerHub*. URL: `https://hub.docker.com/_/mysql`.

[6] *PHPMyAdmin image documentation in DockerHub*. URL: `https://hub.docker.com/_/phpmyadmin`.

[7] *Python smtplib documentation*. URL: `https://docs.python.org/es/3/library/smtplib.html`.

[8] *Traefik image documentation in DockerHub*. URL: `https://hub.docker.com/_/traefik`.

[9] *Traefik official documentation*. URL: `https://doc.traefik.io/traefik/`.