Documentación Ejercicio 4:

1. Introducción

Este documento técnico describe la implementación de una arquitectura contenerizada para una aplicación web de gestión de tareas (tipo *TODO list*, es decir, una lista de tareas pendientes) desarrollada en **Node.js** usando el framework **Express**. La solución utiliza **Redis** como base de datos en memoria para almacenar las tareas de forma persistente, y **Nginx** como **proxy inverso** y **balanceador de carga** al frente de múltiples instancias de la aplicación Node.js. Todo el conjunto de servicios se orquesta mediante **Docker** y **Docker Compose**, lo que permite ejecutar la aplicación de manera aislada, reproducible y escalable en contenedores.

En las siguientes secciones se detallan la arquitectura implementada, las decisiones técnicas tomadas durante su diseño, un diagrama ASCII que ilustra la comunicación entre los servicios, un análisis de las ventajas y desventajas de esta solución, y propuestas de mejora orientadas a un entorno de producción. El estilo es formal y técnico, organizando la información en secciones claras para facilitar su lectura y comprensión.

2. Arquitectura Implementada

Descripción general: La arquitectura está compuesta por varios contenedores Docker, cada uno cumpliendo un rol específico dentro de la aplicación de tareas. En esencia, se han separado las responsabilidades en tres capas principales: la capa de **balanceo de carga** (Nginx), la capa de **aplicación web** (Node.js/Express) con múltiples instancias para atender concurrentemente las solicitudes, y la capa de **base de datos** (Redis) para el almacenamiento de las tareas. Estos componentes se comunican entre sí en una red interna definida por Docker Compose, mientras que sólo el servicio de Nginx expone un puerto al host para acceso de los usuarios.

Componentes de la solución:

- Nginx (Reverse Proxy/Load Balancer): Un contenedor basado en la imagen oficial de Nginx, actúa también como balanceador de carga. Este contenedor es el único expuesto directamente al exterior; es decir, los clientes interactúan con Nginx, el cual a su vez se comunica con los contenedores de la aplicación interna.
- Aplicación Node.js (Instancias Web): Varias instancias del servidor web de la aplicación, cada una ejecutándose en su propio contenedor Docker con Node.js y Express. Todas las instancias ejecutan el mismo código de la aplicación de tareas (servidor Express que maneja las operaciones de la lista TODO) y escuchan en el mismo puerto interno. Docker Compose permite escalar este servicio horizontalmente lanzando múltiples contenedores idénticos.
- Base de datos Redis (con datos persistentes):
  En este proyecto usamos Redis como base de datos para guardar las tareas. Redis

funciona en memoria, lo que lo hace muy rápido. Para no perder la información cuando se apaga o reinicia el contenedor, configuramos Redis con persistencia: guarda copias de los datos en el disco (dentro de la carpeta /data del contenedor).

Además, usamos un volumen de Docker para que esa carpeta /data esté conectada al disco de nuestra máquina. Así, aunque borremos o reiniciemos el contenedor, los datos no se pierden.

- Orquestación con Docker Compose: El archivo docker-compose.yml define los tres servicios mencionados (nginx, app(aplicación Node.js) y redis), incluyendo configuraciones como imágenes/base de contenedor, puertos, dependencias y volúmenes. Al ejecutar docker-compose up, se lanza toda la arquitectura de manera coordinada. Compose crea una red de aplicación única donde cada contenedor puede referenciar a los otros por nombre de servicio.

En resumen, la arquitectura implementada separa claramente la funcionalidad de cada capa en contenedores distintos, manteniendo un acoplamiento débil entre servicios y facilitando la escalabilidad horizontal. A continuación, se explican las principales decisiones técnicas que llevaron a este diseño.

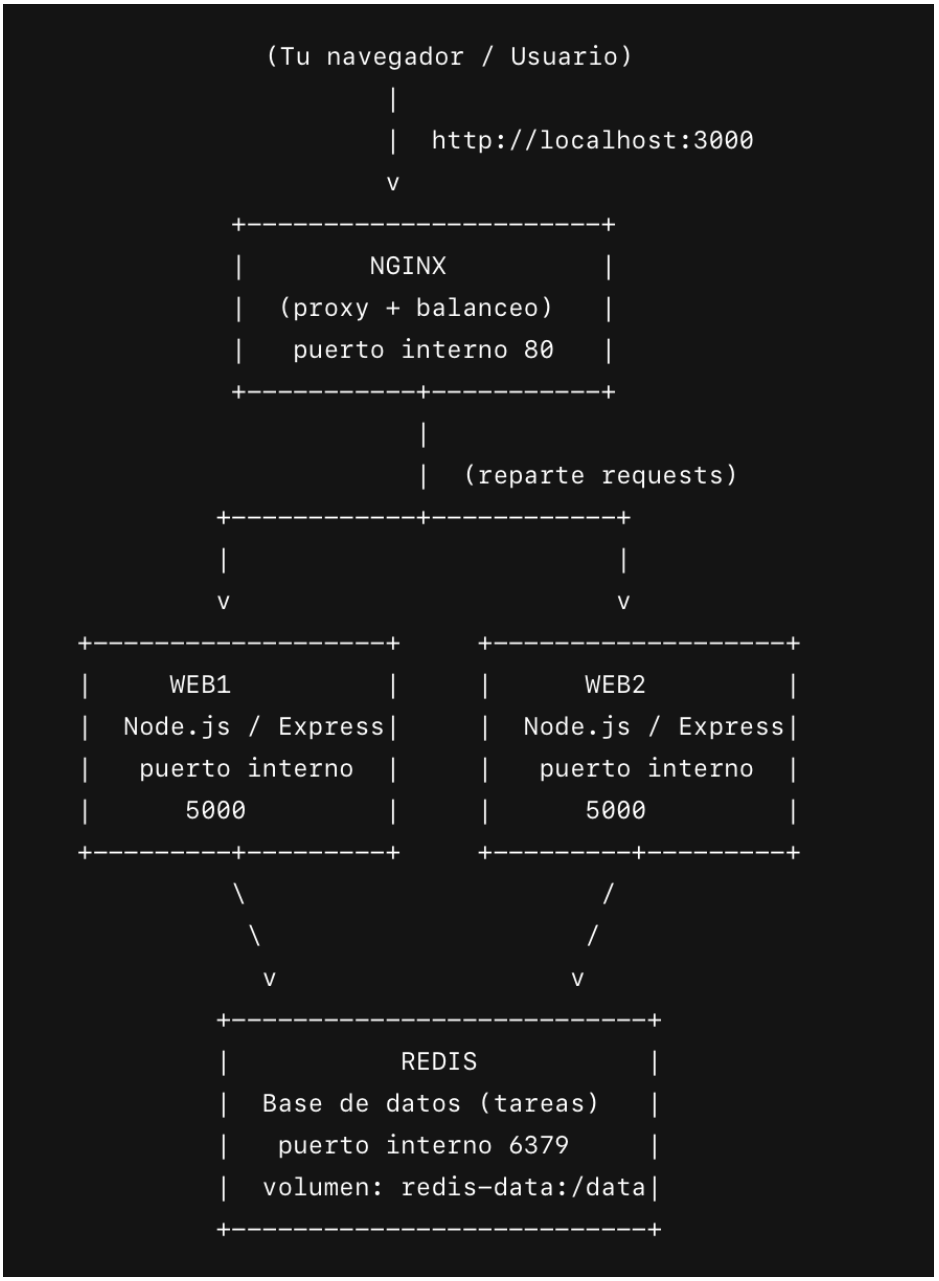
3. Decisiones Técnicas

Para construir esta aplicación, tomamos algunas decisiones clave que nos ayudaron a que el sistema sea más organizado, fácil de escalar y mantener:

- Separación de componentes: Dividimos la app en tres partes: el servidor web (Nginx), la aplicación (Node.js) y la base de datos (Redis). Cada uno corre en su propio contenedor. Esto hace que sea más fácil controlar cada parte por separado, agregar o quitar instancias cuando haga falta y solucionar problemas sin afectar a todo el sistema.
- Uso de Nginx para balancear la carga: Usamos Nginx para que actúe como entrada principal al sistema. Se encarga de recibir todas las solicitudes del usuario y repartirlas entre varias copias (instancias) de la app Node.js. Esto permite que más usuarios puedan usar la app al mismo tiempo sin que se vuelva lenta.
- Redis como base de datos rápida y compartida: Elegimos Redis porque es muy rápido para guardar y leer tareas. Además, al ser compartida por todas las instancias de la app, garantiza que los datos estén siempre sincronizados. Redis guarda los datos en el disco para no perder nada si el contenedor se apaga.
- Uso de Docker Compose: Para simplificar todo, usamos Docker Compose. Así definimos todos los servicios y cómo se conectan en un solo archivo. Esto hace que levantar el sistema sea tan fácil como correr un comando, y que cualquier desarrollador pueda tener el mismo entorno de trabajo sin complicaciones.

4. Diagrama de Comunicación

A continuación se muestra un **diagrama ASCII** que representa la arquitectura y el flujo de comunicación entre los servicios (*declaración de autoría: Se utilizó la ayuda de chat GPT para poder representar en formato ASCII el diagrama):*



El diagrama ilustra claramente la **separación de responsabilidades**: Nginx gestiona la entrada de tráfico y el reparto de carga, las instancias Node.js procesan la lógica de la

aplicación (atender solicitudes, gestionar tareas) y delegan el almacenamiento a Redis, y Redis almacena los datos de forma centralizada para todas las instancias. Esta comunicación encadenada asegura que el usuario obtenga una respuesta como si fuese un solo servidor unificado, cuando en realidad detrás hay múltiples procesos cooperando.

5. Análisis de Ventajas y Desventajas

Ventajas

- **Escalabilidad y mejor respuesta ante carga:** al contar con dos instancias de la aplicación (web1 y web2), Nginx puede distribuir las solicitudes entre ambas. Esto reduce la posibilidad de saturar un único proceso y permite soportar más usuarios al mismo tiempo sin cambiar el código.
- **Separación clara de responsabilidades:** cada componente cumple un rol específico (Nginx como punto de entrada y balanceador, Node.js como lógica de negocio, Redis como almacenamiento). Esta separación hace que el sistema sea más ordenado y facilita comprender “qué hace cada parte”.
- **Facilidad de mantenimiento y cambios:** al estar desacoplados, es posible actualizar o reiniciar un servicio sin necesariamente afectar a los demás. Por ejemplo, se puede reconstruir la imagen de la aplicación web sin modificar Redis o la configuración del balanceador.
- **Entorno reproducible:** Docker y Docker Compose permiten que el entorno sea consistente para cualquier integrante del equipo. Con un solo comando se levanta la misma arquitectura, reduciendo problemas típicos de configuración (“en mi máquina funciona”).
- **Punto de acceso único y controlado:** el sistema se accede únicamente mediante http://localhost:3000, que apunta a Nginx. Esto simplifica el uso y permite centralizar reglas (por ejemplo, futuras configuraciones de seguridad, compresión, redirecciones, etc.) sin tocar la aplicación.
- **Persistencia de datos:** Redis utiliza un volumen (redis-data) para guardar la información en disco. En consecuencia, si se detiene y vuelve a iniciarse el entorno, las tareas almacenadas se mantienen, lo cual valida la persistencia solicitada en la consigna.

Desventajas

- **Mayor complejidad operativa:** en comparación con ejecutar una única aplicación localmente, esta solución requiere gestionar varios servicios (nginx, dos instancias web y redis) y comprender la comunicación entre ellos (red interna, puertos y dependencias).
- **Puntos únicos de fallo en esta versión:** si Nginx se detiene, los usuarios no pueden acceder al sistema porque es el único punto de entrada. De forma similar, si

Redis no está disponible, la aplicación no podrá recuperar ni guardar tareas correctamente.
- **Mayor consumo de recursos:** levantar múltiples contenedores incrementa el uso de memoria y CPU, especialmente al tener más de una instancia de Node.js. Para entornos con recursos limitados, esto puede ser un costo relevante.

En conclusión, la arquitectura implementada aporta beneficios claros en escalabilidad, organización y consistencia del entorno, a cambio de introducir más componentes y complejidad de operación. Esto es esperable en sistemas distribuidos y se alinea con los objetivos del obligatorio (contenedores, redes internas, persistencia y balanceo).

6. Propuestas de mejora para un entorno de producción

(Se utilizó la ayuda de la inteligencia artificial para entender qué mejoras se pueden hacer en entorno de producción)

Si bien la arquitectura actual es adecuada para desarrollo y pruebas, en un entorno de producción sería necesario aplicar algunas mejoras para reforzar la **seguridad**, la **disponibilidad** y la **operación** del sistema.

En primer lugar, sería recomendable incorporar **HTTPS** y mejorar la seguridad de red. Esto implicaría configurar Nginx con certificados SSL/TLS válidos, exponer únicamente los puertos necesarios (por ejemplo, 80/443 en Nginx) y mantener los puertos internos de la aplicación y Redis accesibles solo dentro de la red de Docker. También podría habilitarse autenticación en Redis y, si fuera necesario, usar reglas adicionales de firewall o redes separadas para aislar mejor los componentes.

En cuanto a alta disponibilidad, una evolución natural sería pasar de Docker Compose a un orquestador más avanzado como **Docker Swarm** o **Kubernetes**, que permiten definir réplicas de servicios, reinicios automáticos ante fallos y escalado dinámico según la carga. De esta forma se podrían mitigar algunos puntos únicos de fallo (por ejemplo, añadiendo réplicas de Nginx o configuraciones de Redis con alta disponibilidad).

7. Conclusión

En este documento se presentó la arquitectura contenerizada de una aplicación web de gestión de tareas basada en **Node.js**, **Redis** y **Nginx**, orquestada mediante **Docker Compose**. La solución implementada demuestra cómo dividir la aplicación en servicios especializados (balanceador/proxy, lógica de negocio y base de datos) permite lograr un sistema más modular, escalable y alineado con las prácticas modernas de desarrollo.

En resumen, la arquitectura implementada constituye una base sólida para ejecutar la aplicación de tareas en contenedores. Con las mejoras sugeridas, puede evolucionar hacia una solución robusta y preparada para producción, capaz de crecer en cantidad de usuarios y funcionalidades manteniendo estabilidad y mantenibilidad.