Proyecto despliegue de aplicación con contenedores: Local y Nube

Integrantes

Alejandro Marin Hoyos
2259353-3743
Carlos Alberto Camacho Castaño
2160331-3743
Harrison Ineey Valencia Otero
2159979-3743
Kevin Alexander Marin
2160364-3743
Manuel Antonio Vidales Duran
2155481-3743
Yessica Fernanda Villa Nuñez
2266301-3743

Infraestructuras paralelas y distribuidas

Universidad del Valle
Tuluá, Valle del Cauca
Diciembre 2024

1. Introducción

Este proyecto nace debido a la necesidad de la movilidad económica y eficiente para estudiantes universitarios, este proyecto propone una solución integral basada en una aplicación web intuitiva. La aplicación conecta a propietarios de motocicletas que son los arrendadores, con estudiantes arrendatarios que buscan opciones de transporte accesibles, creando un puente confiable entre ambas partes.

El enfoque principal de este proyecto es lograr un buen despliegue de la aplicación en entornos locales y en la nube, garantizando escalabilidad del servidor, flexibilidad y altos estándares de satisfacción del cliente. Utilizando tecnologías modernas como Docker Compose y Kubernetes, la solución asegura una arquitectura modular y eficiente que soporta operaciones continuas y robustas en ambas plataformas.

Objetivo General:

- Diseñar, desarrollar y desplegar la aplicación web modular y escalable para facilitar el transporte económico de estudiantes universitarios. La solución integrará Backend, Base de Datos y Frontend mediante una arquitectura distribuida, utilizando contenedores y servicios en la nube para garantizar disponibilidad, escalabilidad y flexibilidad operativa, optimizando la experiencia del usuario y manteniendo altos estándares de satisfacción.

objetivos específicos:

- Implementar una arquitectura distribuida y modular que integre Backend, Base de Datos y Frontend, asegurando comunicación eficiente a través de una API REST y habilitando operaciones CRUD.
- Desplegar la solución en entornos locales y en la nube, utilizando Docker Compose para el desarrollo local y Kubernetes para un entorno de producción escalable y de alta disponibilidad.
- Optimizar la experiencia de usuario mediante el diseño de una interfaz web intuitiva y accesible, acompañada de funcionalidades clave como login, geolocalización, historial de viajes y filtros personalizados, garantizando la usabilidad y satisfacción de los usuarios.

2. Solución Local

Descripción de la arquitectura local

La arquitectura local emplea una solución basada en contenedores, donde cada componente del sistema está aislado en su propio contenedor. Esto asegura modularidad, facilidad de mantenimiento y portabilidad. Los tres componentes principales son:

1. **Backend:** Implementado como un servicio que gestiona la lógica de negocio y se conecta a la base de datos.
2. **Base de Datos:** Un servicio dedicado para almacenar y gestionar los datos persistentes del sistema.
3. **Frontend:** Una interfaz de usuario desarrollada para interactuar con el backend, presentada al cliente.

Detalle de la Arquitectura Local

El diseño modular se logra mediante la separación de responsabilidades en componentes independientes:

- **Backend:** Construido con Node.js/Express.
- **Base de Datos:** Utiliza un motor como MySQL, configurado para persistir datos en un volumen local.
- **Frontend:** Creado con Angular y desplegado como un servicio estático en un servidor web, como Nginx.

Configuración de los contenedores e interacciones

Backend: Este `Dockerfile` configura un contenedor ligero para el backend usando la imagen base `node:18-alpine`. Define el directorio de trabajo `/usr/src/app`, copia los archivos `package.json` para instalar dependencias con `npm install`, y luego transfiere todo el código fuente al contenedor. Expone el puerto `9001` para la comunicación externa y utiliza el comando `npm start` para arrancar la aplicación al iniciar el contenedor. Es eficiente, modular y sigue buenas prácticas para proyectos Node.js.

```dockerfile
# Use Node.js base image
FROM node:18-alpine

# Create app directory
WORKDIR /usr/src/app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy app source code
COPY . .

# Expose port
EXPOSE 9001

# Command to run the app
CMD ["npm", "start"]
```

Imagen Dockerfile Backend

Base de datos: En la configuración de la base de datos, se creó un script SQL que permite la generación automática de las tablas necesarias para el proyecto al iniciar el contenedor. Este enfoque asegura que la estructura de la base de datos esté siempre preparada y lista para su uso sin necesidad de configuraciones manuales adicionales.

Frontend: Este Dockerfile optimiza el despliegue del frontend al separar las etapas de construcción y producción. La etapa de construcción instala dependencias y genera los archivos estáticos con Node.js, mientras que la etapa de producción utiliza Nginx para servirlos de manera eficiente. Esta estrategia minimiza el tamaño del contenedor final y mejora el rendimiento.

```dockerfile
# Build stage
FROM node:18-alpine as build

# Set working directory
WORKDIR /usr/src/app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy app source
COPY . .

# Build the app
RUN npm run build

# Production stage
FROM nginx:alpine

# Copy built assets from build stage
COPY --from=build /usr/src/app/dist/*/browser /usr/share/nginx/html

# Copy nginx configuration
COPY nginx.conf /etc/nginx/conf.d/default.conf

# Expose port
EXPOSE 80
```

Imagen Dockerfile Fronted

Configuración del Docker Compose:

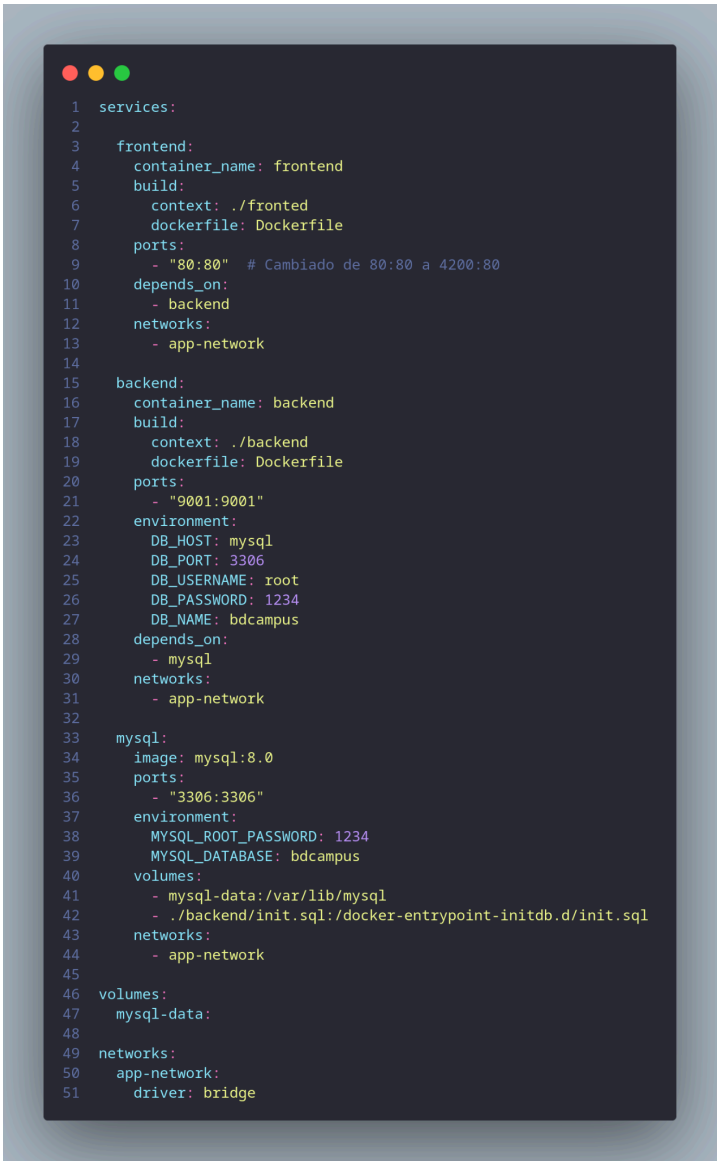
Este archivo docker-compose.yml define la arquitectura de un entorno local con tres servicios interconectados (frontend, backend y base de datos MySQL) dentro de una red común app-network.

Frontend: Construido desde el directorio ./fronted usando su Dockerfile. Expone el puerto 80 para servir la aplicación en producción y depende del backend para estar operativo.

Backend: Construido desde el directorio ./backend, expone el puerto 9001 y se configura con variables de entorno para conectarse al servicio MySQL. También depende de la base de datos para funcionar.

MySQL: Usa la imagen oficial de MySQL 8.0, crea automáticamente la base de datos bdcampus y ejecuta un script de inicialización init.sql. Los datos se persisten con el volumen mysql-data.

Todos los servicios comparten la red app-network para facilitar su comunicación.



```yaml
services:

  frontend:
    container_name: frontend
    build:
      context: ./fronted
      dockerfile: Dockerfile
    ports:
      - "80:80"  # Cambiado de 80:80 a 4200:80
    depends_on:
      - backend
    networks:
      - app-network

  backend:
    container_name: backend
    build:
      context: ./backend
      dockerfile: Dockerfile
    ports:
      - "9001:9001"
    environment:
      DB_HOST: mysql
      DB_PORT: 3306
      DB_USERNAME: root
      DB_PASSWORD: 1234
      DB_NAME: bdcampus
    depends_on:
      - mysql
    networks:
      - app-network

  mysql:
    image: mysql:8.0
    ports:
      - "3306:3306"
    environment:
      MYSQL_ROOT_PASSWORD: 1234
      MYSQL_DATABASE: bdcampus
    volumes:
      - mysql-data:/var/lib/mysql
      - ./backend/init.sql:/docker-entrypoint-initdb.d/init.sql
    networks:
      - app-network

volumes:
  mysql-data:

networks:
  app-network:
    driver: bridge
```

Imagen Docker Compose

Comandos para desplegar la solución local:

```
Unset
# Ingresar a Docker Hub y logearse con tus credenciales
docker login

# Detener todos los contenedores en ejecución y los detenidos
docker stop $(docker ps -q -a)

# Eliminar contenedores, redes, imágenes no utilizadas y volúmenes para liberar espacio
docker system prune -a

# Construir las imágenes de los servicios definidos en el archivo docker-compose.yml
docker compose build

# Levantar los contenedores definidos en el archivo docker-compose.yml
docker compose up

# Nota: La base de datos se crea al ejecutar 'docker compose up', pero es necesario reiniciar el contenedor de backend para que la base de datos se inicialice correctamente.

# Detener y eliminar todos los contenedores, redes y volúmenes asociados al proyecto
docker compose down -v

# Reiniciar el contenedor específico de backend, reemplaza 'name-image-backend' por el nombre del servicio o contenedor correspondiente
docker compose restart name-image-backend
```

3. Solución en la Nube

Replicación de la Arquitectura Local en la Nube

Para replicar la arquitectura local en la nube, utilizamos DigitalOcean, aprovechando una clave de acceso (key) para gestionar de manera segura las instancias y ejecutar los comandos necesarios a través de la consola. La misma estructura de contenedores Docker utilizada en el entorno local se desplegó en la nube, garantizando que los servicios de frontend, backend y base de datos funcionaran de manera similar en ambos entornos.

Configuración del Balanceador de Carga

Se configuró un balanceador de carga para el backend, replicando el servicio de backend tres veces para distribuir el tráfico entrante entre las instancias y mejorar la disponibilidad y escalabilidad. Este balanceador de carga garantiza que las solicitudes de los usuarios se distribuyan de manera equitativa entre los contenedores backend, optimizando el rendimiento y evitando la sobrecarga de una única instancia.

Descripción de las Configuraciones Utilizadas para Garantizar la Disponibilidad

Para asegurar que la aplicación esté siempre disponible, se utilizaron configuraciones específicas en DigitalOcean que incluyen:

- **Escalabilidad automática:** Se configuraron instancias para que pudieran escalar según la demanda de tráfico.
- **Réplicas del backend:** Tres instancias del backend corren simultáneamente para asegurar que si alguna falla, el tráfico sea redirigido automáticamente a las otras instancias.
- **Balanceo de carga:** Un balanceador de carga distribuye las solicitudes entre las instancias backend para maximizar la eficiencia y minimizar tiempos de inactividad.

Uso de Servicios en la Nube

En DigitalOcean, se configuraron varios servicios clave para garantizar la operación eficiente de la aplicación:

- **Redes:** Se estableció una red privada para permitir la comunicación segura entre los servicios de frontend, backend y base de datos, asegurando que no haya tráfico externo innecesario.
- **Seguridad:** Se configuraron reglas de firewall para restringir el acceso a las instancias y permitir solo el tráfico necesario (por ejemplo, puertos 80 y 9091 para HTTP y HTTPS, respectivamente). También se utilizó autenticación basada en claves SSH para acceder de forma segura a las instancias.
- **Almacenamiento y Volúmenes:** Se configuraron volúmenes persistentes en DigitalOcean para almacenar los datos de la base de datos y garantizar que la información se mantenga intacta, incluso si los contenedores se reinician.
- **Flujo de Despliegue en la Nube**
- El flujo de despliegue en DigitalOcean fue realizado de manera secuencial, a través de la consola, usando los siguientes pasos:

Conexión a la instancia de DigitalOcean mediante SSH utilizando la clave privada generada.

```
ssh -i /Descargas/key.pem user@206.189.190.150
```

Instalación de Docker y Docker Compose en la instancia para gestionar los contenedores.

```
sudo apt-get update
```

```
sudo apt-get install docker.io
```

```
sudo apt-get install docker-compose
```

Transferencia del código fuente de la aplicación a la instancia usando `scp` o Git.

```
scp -i /Descargas/key.pem -r /Descargas/ProyectoDocker user@206.189.190.150:/Descargas/ProyectoDocker
```

Construcción de los contenedores en la instancia de DigitalOcean utilizando docker compose

```
docker-compose build
```

Despliegue de la aplicación con Docker Compose, levantando los contenedores de frontend, backend y base de datos.

```
docker-compose up -d
```

Configuración del balanceador de carga en DigitalOcean para distribuir el tráfico entre las réplicas del backend.

Verificación de la correcta operación de la aplicación mediante la navegación a la IP pública del droplet o el dominio configurado.

4. Análisis y Conclusiones

Análisis del Rendimiento de la Aplicación en Ambiente Local y en la Nube

1. Ambiente Local:

- **Velocidad:** La latencia es baja en redes internas, ideal para operaciones locales.
- **Capacidad:** Depende del hardware instalado, lo que puede limitar el rendimiento en picos de uso.
- **Optimización:** Puede configurarse específicamente para las necesidades de la aplicación.
- **Riesgo de Caídas:** Aumenta si el hardware falla o no hay redundancia.

2. Ambiente en la Nube:

- **Velocidad:** Puede tener mayor latencia si los servidores están lejos de los usuarios.
- **Capacidad:** Escalable dinámicamente para manejar picos de demanda sin pérdida de rendimiento.
- **Optimización:** Ofrece herramientas avanzadas para monitoreo y ajuste continuo.
- **Disponibilidad:** Alta redundancia y tolerancia a fallos integradas por el proveedor.

Analizar la latencia, escalabilidad y disponibilidad del sistema.

1. Latencia:

Análisis:

Si la latencia es un problema, podría estar relacionada con la distancia entre los usuarios y los servidores o con cuellos de botella en el tráfico de red.

La baja utilización del CPU y del disco indica que los recursos no están saturados, lo que sugiere que la latencia actual es aceptable para la carga presente.

Recomendaciones:

Implementar Content Delivery Networks (CDNs) para mejorar la latencia en usuarios geográficamente distantes.

Monitorear tiempos de respuesta en tiempo real para identificar y resolver problemas de latencia más rápidamente.

2. Escalabilidad:

Análisis:

El sistema está preparado para escalar horizontalmente (agregar más instancias) o verticalmente (aumentar recursos de cada servidor) si fuera necesario.

La naturaleza de los picos (simultáneos en CPU, ancho de banda y disco) sugiere que la escalabilidad debe ser equilibrada en todos los recursos.

Recomendaciones:

Adoptar infraestructura basada en la nube para escalar dinámicamente según la demanda.

Realizar pruebas de carga para determinar el punto exacto en el que los recursos comienzan a saturarse.

3. Disponibilidad:

- **Análisis:**

La disponibilidad parece estar garantizada bajo la carga actual, pero un incremento abrupto podría poner en riesgo la capacidad de respuesta del sistema si no está bien diseñado.

En un entorno local, la disponibilidad depende de la infraestructura física; en la nube, se pueden aprovechar opciones como balanceadores de carga y configuraciones de alta disponibilidad.

- **Recomendaciones:**

Implementar balanceo de carga y redundancia para garantizar disponibilidad en caso de fallos.

Usar monitoreo continuo para detectar y resolver problemas antes de que impacten la disponibilidad.

Identificación de retos y soluciones

- **Rendimiento**

Desafío: Sitio web lento
Solución: Comprimir archivos y optimizar imágenes

- **Seguridad**

Desafío: Vulnerabilidades
Solución: HTTPS, validación de datos, actualizaciones regulares

- **Base de Datos**

Desafío: Lentitud en consultas
Solución: Optimización de queries, caché, backups

- **Mantenimiento**

Desafío: Actualizaciones complicadas
Solución: Automatización, documentación clara

Reflexión sobre las tecnologías

- Docker brilla por su capacidad de crear entornos consistentes y portables. Facilita enormemente el desarrollo al eliminar el clásico problema de "en mi máquina funciona". También permite un mejor aprovechamiento de recursos comparado con máquinas virtuales tradicionales
- Kubernetes, por su parte, es excelente para orquestar contenedores a gran escala. Sus principales ventajas son el escalado automático, la alta disponibilidad y la gestión eficiente de recursos.
- Las tecnologías de contenedores y orquestación tienen beneficios claros: facilitan el desarrollo con entornos consistentes, permiten escalar fácilmente según la demanda y reducen costos operativos a largo plazo. La entrega de aplicaciones es más rápida y el control de versiones más eficiente.

Limitaciones: requieren una importante inversión inicial en capacitación, tienen una curva de aprendizaje pronunciada y pueden complicar la arquitectura del sistema. La gestión de datos y la depuración de problemas también pueden ser más complejas.

Elementos clave del análisis

- **Escalabilidad:**

  La escalabilidad, en el despliegue local y en la nube de una página web, implica la capacidad de responder a aumentos en la carga de usuarios o tráfico. Docker facilita esto al permitir la creación y gestión de múltiples instancias de contenedores, pero es igual de importante considerar herramientas complementarias, como balanceadores de carga y escalado automático en plataformas en la nube. Esto asegura que la infraestructura pueda crecer o reducirse según las necesidades.

- **Fiabilidad:**

  Un sistema fiable garantiza que la página web esté operativa bajo distintas condiciones, ya sea en entornos locales o en la nube. Docker contribuye al aislamiento y la consistencia entre entornos, mientras que servicios de monitoreo, recuperación ante fallos y arquitecturas redundantes, como clústeres distribuidos, aseguran que el sistema pueda tolerar errores y seguir funcionando sin interrupciones significativas.

- **Costo:**

  El costo en este contexto incluye tanto los gastos iniciales, como servidores o licencias, como los costos recurrentes en la nube, como uso de instancias y almacenamiento. Docker puede optimizar los costos al reducir el desperdicio de recursos mediante contenedores ligeros, pero también es importante evaluar precios de servicios en la nube, costos de transferencia de datos y cualquier dependencia externa para mantener la sostenibilidad económica.

- **Optimización de recursos:**

  El uso eficiente de recursos es clave para maximizar el rendimiento de la página web. Docker ayuda a empaquetar aplicaciones de manera ligera, pero la optimización también incluye elegir hardware adecuado, ajustar configuraciones del servidor web y emplear mecanismos de cacheo como Redis o CDNs. En la nube, seleccionar el tipo correcto de instancia y optimizar los tiempos de ejecución son esenciales para reducir costos sin sacrificar desempeño.

- **Seguridad:**

  La seguridad abarca la protección de datos y la prevención de ataques en todo el sistema. Esto incluye políticas como la encriptación de datos, tanto en tránsito como en reposo, y la implementación de controles de acceso. Además, en despliegues con Docker, es crucial seguir prácticas seguras, como usar imágenes verificadas y mantener el sistema operativo actualizado. En la nube, la seguridad puede ser reforzada mediante cortafuegos, monitoreo de tráfico y gestión adecuada de credenciales y claves.

Conclusión

El proyecto permitió identificar áreas clave para mejorar la escalabilidad, la fiabilidad y la seguridad en el desarrollo y despliegue de aplicaciones, destacando el papel fundamental de Docker como herramienta para lograr despliegues eficientes y consistentes tanto en entornos locales como en la nube. Además, el proceso evidenció la necesidad de integrar prácticas como el monitoreo continuo, el uso de arquitecturas escalables y la implementación de medidas robustas de seguridad. La experiencia obtenida subraya la importancia de una planificación integral que contemple no solo las necesidades actuales, sino también la capacidad de adaptación a futuras demandas. Esto refuerza el valor de adoptar tecnologías modernas y estrategias sólidas para garantizar el éxito sostenible de este proyecto tecnológico.