Proyecto Final Infraestructuras paralelas y distribuidas

Drive Now

Integrantes:

Carlos Stiven Ruiz Rojas - 2259629
Jhony Fernando Duque Villada - 2259398
Jairo Andres Gomez Cardona - 2259332
Juan Camilo Diaz Valencia - 2259583
Juan David Rojas Narvaez - 2259673

Docente
Carlos Andres Delgado Saavedra

Universidad del Valle - Seccional Tuluá

2024

Universidad del Valle

Introducción:

Este proyecto es un servicio de alquiler de vehículos, llamado "Drive Now". La plataforma está diseñada para facilitar la búsqueda, visualización y reserva de vehículos para los usuarios. Los vehículos están categorizados (como carros, camionetas y motos), y cada uno muestra detalles clave como marca, modelo, color, precio de alquiler por día, etc. Además, la página cuenta con funcionalidades básicas de navegación, como un menú con opciones de "Inicio", "Sobre Nosotros", "Servicios", y "Contacto", junto con botones para iniciar sesión o registrarse, lo que indica un enfoque en la personalización y seguridad del usuario.

Objetivos:

El objetivo de dockerizar este proyecto tiene varios fines, entre ellos puedes estar:

- Ofrecer portabilidad y facilitar a la hora de hacer el despliegue.
- Nos evitaremos problemas de configuración a futuro ya que se puede aislar entornos de desarrollo.
- Si llega a haber un inconveniente se puede solucionar sin afectar a los demás.

Solución local usando Docker-Compose

I. El proyecto utiliza Docker Compose para gestionar de manera sencilla la ejecución de múltiples contenedores en un entorno local. Esta arquitectura cuenta con tres componentes principales: backend, frontend y base de datos (PostgreSQL). Docker Compose ejecuta estos servicios, los conecta a través de una red interna, y maneja su configuración para que todos trabajen de manera conjunta y eficiente.

- **Red de contenedores:**

  Los tres servicios (backend, frontend y database) están conectados a una red compartida llamada backend_network, lo que les permite comunicarse entre sí de manera segura y eficiente. Esta red interna facilita que los servicios interactúen sin depender de la exposición pública de puertos innecesarios.

- **Contenedores:**
  En el archivo docker-compose se definen tres contenedores:

- Contenedor de la base de datos (PostgreSQL): Este servicio ejecuta una base de datos PostgreSQL que almacena los datos de la aplicación, como la información de los vehículos, usuarios, etc.

- Contenedor del backend: Este contenedor ejecuta el backend de la aplicación, que está basado en Node.js con TypeScript. Proporciona una API RESTful que interactúa con la base de datos y responde a las solicitudes del frontend.

- Contenedor del frontend: Este contenedor ejecuta el frontend de la aplicación, que está servido a través de Nginx. Contiene archivos estáticos construidos con React, que se entregan a los usuarios finales.

II. El diseño de la arquitectura se basa en la separación de responsabilidades, lo que significa que cada servicio se encarga de una parte específica del sistema, manteniendo la modularidad y facilitando el mantenimiento.

- **Base de datos(PostgreSQL):**

  - Configuración: Se utiliza una imagen oficial de PostgreSQL (postgres:15), configurada con variables de entorno para establecer el nombre de usuario, la contraseña y la base de datos predeterminada.
  - Persistencia: Se usa un volumen (postgres_data) para asegurar que los datos sean persistentes incluso si el contenedor de la base de datos se reinicia.

- **Backend(Node.js con TypeScript):**

  - Función: Proporciona la lógica de negocio y la API RESTful que interactúa con la base de datos y envía respuestas al frontend.

  - Configuración: Este contenedor utiliza el Dockerfile ubicado en Drive-Now para construir el backend. El contenedor expone el puerto 3000 y depende de la base de datos, lo que significa que se asegura de que el contenedor de la base de datos esté listo antes de intentar iniciar el backend.

  - Interacción: El backend se comunica con la base de datos PostgreSQL para realizar operaciones CRUD (crear, leer, actualizar, eliminar) sobre los datos.

- **Frontend(Nginx):**

  - Función: Presenta la interfaz de usuario para interactuar con la aplicación de alquiler de vehículos. Realiza peticiones al backend para obtener los datos y mostrarlos al usuario.

- Configuración: Este contenedor utiliza el Dockerfile de la carpeta DriveNowFrontend para construir y servir la aplicación frontend. Se expone el puerto 8080 en el contenedor, lo que significa que el frontend estará disponible en localhost:8080 para el usuario.
- Interacción: El frontend realiza peticiones HTTP (como GET, POST, PUT, DELETE) al backend para obtener datos o realizar acciones en la base de datos.

III. Cada servicio está definido con su propio contenedor Docker y configurado para interactuar con los otros a través de la red interna definida en el archivo docker-compose.yml.

- **Base de datos(PostgreSQL)**

  - Dockerfile y configuración:
    La base de datos está configurada con la imagen oficial postgres:15. Se define el nombre de usuario (POSTGRES_USER), la contraseña (POSTGRES_PASSWORD) y el nombre de la base de datos (POSTGRES_DB), lo que permite inicializar la base de datos con credenciales específicas.

  - Persistencia de datos:
    Se utiliza un volumen llamado postgres_data para mantener los datos de la base de datos persistentes, de modo que, incluso si el contenedor de la base de datos se reinicia, los datos no se pierdan.

- **Backend (Node.js con TypeScript)**

  - Dockerfile y Configuración:
    El contenedor del backend utiliza el Dockerfile en la carpeta Drive-Now. Primero instala las dependencias usando npm install, luego copia los archivos de la aplicación y expone el puerto 3000.

  - Interacción con la Base de Datos:
    El contenedor del backend depende del contenedor de la base de datos, lo que asegura que el servicio de base de datos esté disponible antes de iniciar el backend. El backend se conecta a la base de datos usando las credenciales proporcionadas en el archivo .env y realiza consultas para obtener o modificar datos.

  - Interacción con el Frontend:
    El contenedor del frontend realiza solicitudes HTTP al contenedor del backend para interactuar con los datos almacenados en la base de datos. Estas

solicitudes pueden incluir obtener la lista de vehículos, agregar nuevos vehículos, o actualizar los detalles de los vehículos.

- **Frontend (Nginx)**

  - Dockerfile y Configuración:
    El contenedor frontend usa dos etapas en su Dockerfile. La primera es la etapa de construcción, donde se instalan las dependencias del frontend y se genera la versión de producción de la aplicación (por ejemplo, usando npm run build). La segunda etapa usa la imagen de Nginx (nginx:1.21-alpine) para servir los archivos estáticos generados en la primera etapa.

  - Interacción con el Backend:
    El frontend realiza peticiones HTTP al backend a través del contenedor del backend. Por ejemplo, puede obtener datos sobre los vehículos a través de una API RESTful proporcionada por el backend.

- Dockerfile frontend:

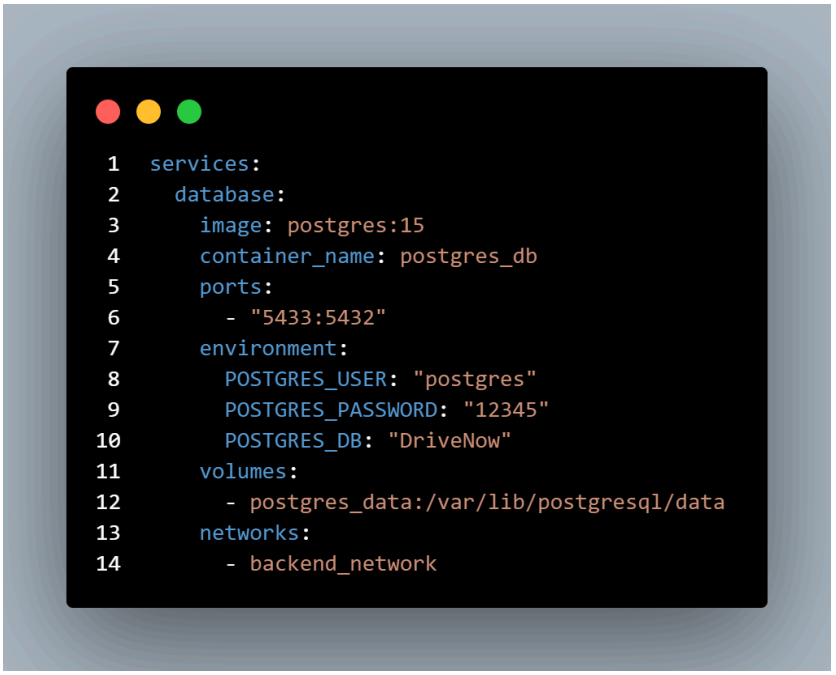
```dockerfile
# Etapa de construcción
FROM node:18 AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Etapa de producción
FROM nginx:1.21-alpine
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]

```

- Dockerfile backend:

```dockerfile
# Usar una imagen base de Node.js
FROM node:18

# Establecer el directorio de trabajo dentro del contenedor
WORKDIR /usr/src/app

# Copiar el archivo package.json y package-lock.json (si existe)
COPY package*.json ./

# Instalar las dependencias del proyecto
RUN npm install

# Copiar todo el código fuente del proyecto al contenedor
COPY . .

# Exponer el puerto en el que la app va a escuchar
EXPOSE 3000

# Comando por defecto para ejecutar el contenedor
CMD ["npx", "ts-node", "src/main/index.ts"]

```

Configuración con docker-compose:

- **Base de datos:**



```yaml
services:
  database:
    image: postgres:15
    container_name: postgres_db
    ports:
      - "5433:5432"
    environment:
      POSTGRES_USER: "postgres"
      POSTGRES_PASSWORD: "12345"
      POSTGRES_DB: "DriveNow"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - backend_network
```

- **Backend:**

```yaml
backend:
  build:
    context: ./Drive-Now
    dockerfile: Dockerfile
  container_name: drive-now_backend
  ports:
    - "3000:3000"
  env_file:
    - ./Drive-Now/.env
  networks:
    - backend_network
  depends_on:
    - database
```

- **Frontend:**

Universidad del Valle

```yaml
  frontend:
    build:
      context: ./DriveNowFrontend
      dockerfile: Dockerfile
    container_name: drive-now_frontend
    ports:
      - "8080:80"
    networks:
      - backend_network
    depends_on:
      - backend

volumes:
  postgres_data:

networks:
  backend_network:

```

Solución en la nube:

Para llevar a cabo la replicación de este proyecto en la nube, se utilizaron instancias dentro del servicio Google Cloud Run, que fueron desplegadas utilizando como base los contenedores Docker previamente configurados. Este proceso permitió aprovechar las ventajas de la contenedorización y la escalabilidad automática de Google Cloud.

Construcción de Imágenes Docker:

Se inició construyendo las imágenes Docker correspondientes a cada componente del sistema. Específicamente, se crearon imágenes para el backend y el frontend, asegurando que cada una contuviera las dependencias y configuraciones necesarias para su correcto funcionamiento. Estas imágenes se generaron a partir de archivos Dockerfile, que describen cómo empaquetar la aplicación y sus dependencias en un entorno de ejecución portátil y consistente. Este paso es crucial para garantizar que tanto el backend como el frontend puedan ejecutarse de manera idéntica, sin importar el entorno en el que se desplieguen.

1. **Subida de Imágenes al Artifact Registry:**
   Una vez generadas las imágenes Docker, el siguiente paso fue subirlas al Artifact Registry de Google Cloud. Este es un servicio diseñado para almacenar y gestionar imágenes de contenedores de forma segura. Para ello, se utilizó el CLI de Google Cloud, autenticando previamente el entorno para garantizar el acceso al registro. Las imágenes se etiquetan

correctamente con los identificadores correspondientes para backend y frontend, y luego se suben al registro, asegurando su disponibilidad en la nube para los siguientes pasos.

2. **Despliegue en Google Cloud Run:**
   Con las imágenes ya alojadas en el Artifact Registry, se procedió al despliegue de instancias en Google Cloud Run, tanto para el backend como para el frontend. Google Cloud Run permite ejecutar servicios contenedorizados de manera totalmente gestionada, con escalabilidad automática basada en la demanda. Durante este paso, se configuraron las instancias para aceptar solicitudes externas y se asignaron URLs únicas a cada servicio. Esto asegura que cada componente pueda ser accedido de forma independiente y desde cualquier parte del mundo.

3. **Configuración del Balanceo de Carga para el Backend:**
   Para garantizar la disponibilidad y eficiencia del backend, se configuró el balanceo de carga mediante el despliegue de tres instancias de este servicio. Cloud Run se encargó automáticamente de distribuir las solicitudes entrantes entre las instancias activas, permitiendo manejar mayores volúmenes de tráfico sin saturar ninguna de ellas. Además, esta configuración proporciona redundancia, asegurando que si una instancia falla, las demás puedan continuar respondiendo a las solicitudes.

4. **Configuración de la Base de Datos con Google Cloud SQL:**
   La base de datos fue implementada utilizando el servicio Google Cloud SQL, que permite gestionar bases de datos relacionales en la nube de forma segura y eficiente. Durante este paso, se creó una instancia de base de datos y se configuró una dirección IP pública que permitiera la comunicación con el backend. En el backend, se configuró la conexión utilizando esta dirección IP pública, junto con el nombre de host y las credenciales correspondientes. Para facilitar la comunicación, se configuró la base de datos para aceptar conexiones desde la dirección 0.0.0.0/0, que permite el acceso desde cualquier dirección IP. Esto asegura que el backend pueda interactuar con la base de datos desde cualquier origen autorizado.

5. **Configuración de CORS en el Backend:**
   Finalmente, se configuró el backend para manejar adecuadamente las solicitudes del frontend mediante CORS (Cross-Origin Resource Sharing). Esto fue necesario para garantizar que el navegador de los usuarios pudieran comunicarse de manera segura con el backend sin restricciones. Específicamente, se configuró el backend para aceptar solicitudes provenientes de la URL específica del frontend desplegado en Google Cloud Run, evitando conflictos y asegurando una integración fluida entre ambos componentes.

Universidad del Valle

Análisis y conclusiones:

1. Rendimiento en ambiente local vs en la nube:

    - En local, la aplicación funciona rápido y directo porque está en nuestro equipo, pero está limitado dependiendo del equipo que tengamos.

    - En la nube es más fácil aumentar recursos si hace falta, pero puede ser un poco más lento por el internet.

2. Latencia, escalabilidad y disponibilidad:

    - Latencia: En local es más rápida porque no depende de internet y en la nube pueden haber más retrasos según la red.

    - Escalabilidad: La nube es mejor porque podemos agregar más capacidad fácilmente. En el local es más difícil y costoso.

    - Disponibilidad: En la nube es más confiable porque si algo falla, otras partes lo reemplazan, en cambio en local, si algo se daña, toca arreglarlo manualmente.

3. Retos y cómo los resolvemos:

    - Configurar todo fue difícil al principio, pero lo solucionamos buscando guías y probando.

    - La seguridad se solucionó protegiendo los accesos con contraseñas y permisos básicos.

4. Uso de Docker:

    - Docker nos permitió dividir el proyecto en partes (backend, frontend y base de datos) y que cada parte funcionara sin depender de otra.

5. Elementos clave del proyecto:

    - Escalabilidad: Que el sistema aguante más usuarios. la nube lo hace más fácil.

    - Fiabilidad: Que siga funcionando aunque algo falle, lo logramos con réplicas y un balanceador.

- Costo: La implementación en la nube con Google Cloud involucró costos por uso de recursos como máquinas virtuales, balanceadores de carga, almacenamiento y servicios gestionados (bases de datos en la nube). Pero a pesar de estos costos, los beneficios de escalabilidad, redundancia y seguridad justificaron la inversión, especialmente en entornos con alta demanda o necesidad de disponibilidad 24/7.
  Los créditos estudiantiles proporcionados ayudaron a reducir los gastos iniciales, pero se identificó que un uso continuo podría representar un costo significativo sin estos subsidios.

- Optimización: Configuramos límites para que el sistema use sólo lo necesario.

- Seguridad: Restringimos accesos y cuidamos los datos básicos.

Conclusiones:

Docker Compose es una herramienta muy útil para proyectos que necesitan mantener consistencia entre los entornos de desarrollo y producción. Ayuda a evitar problemas de compatibilidad y hace que las configuraciones complicadas sean más sencillas. Esto permite a los desarrolladores concentrarse en mejorar las funciones y la experiencia del usuario, algo clave en una aplicación como una página de alquiler de vehículos. También ayuda a organizar mejor la arquitectura del sistema y mejora su seguridad.

Por su parte, Google Cloud Run tiene grandes ventajas en cuanto a escalabilidad y disponibilidad. Es muy fácil ajustar los recursos para manejar más usuarios cuando hay picos de tráfico, asegurando que el sistema esté siempre disponible. Esto alivia al equipo de preocuparse por la infraestructura y les deja más tiempo para enfocarse en el desarrollo y mejoras del proyecto.

Docker Compose permite un control detallado del entorno, lo que es ideal para personalizaciones específicas. En cambio, Cloud Run simplifica el despliegue y la gestión, ya que solo necesitas subir un contenedor y la nube se encarga del resto. Esto es práctico para equipos que buscan rapidez y menos complicaciones técnicas.

En cuanto a costos, Docker Compose puede ser más económico para entornos pequeños y controlados. Por otro lado, Cloud Run, con su modelo de pago por uso, es perfecto para proyectos que necesitan adaptarse a cambios en la demanda sin gastar demasiado tiempo en administrar servidores.

En resumen, Docker Compose y Google Cloud Run son herramientas que pueden complementarse. Docker es ideal para tener más control en el desarrollo, mientras que Cloud Run es excelente para escalar y facilitar el despliegue. La elección depende de lo que el proyecto necesite y las prioridades del equipo.

https://frontend-25620165595.us-central1.run.app