Informe Proyecto final

Universidad del Valle

Ingeniería en Sistemas

Infraestructuras paralelas y distribuidas

Integrantes:

Juan Pablo Escobar Viveros- 2259519

Edgar Andrés Vargas García - 2259690

Cristian David Rivera Torres - 2259742

Introducción

Breve descripción del proyecto

El proyecto desarrollado consiste en una aplicación web de lista de tareas que permite a los usuarios gestionar sus actividades mediante funcionalidades básicas como visualizar, añadir, editar y eliminar tareas. El enfoque del proyecto radica en desplegar esta aplicación en diferentes entornos utilizando tecnologías modernas como Docker y Docker Compose para contenerizar y gestionar todos los componentes.

La arquitectura de la solución propuesta está diseñada bajo principios de separación de componentes, implementando un backend, una base de datos y un frontend independientes que se comunican entre sí a través de una API REST. Este diseño facilita la escalabilidad, mantenibilidad y flexibilidad del sistema.

Objetivos del proyecto

- Funcionalmente, la aplicación debe permitir a los usuarios realizar operaciones fundamentales para la gestión de tareas, como visualizar una lista de tareas, añadir nuevas tareas, editar tareas existentes y eliminarlas cuando ya no sean necesarias.
- Se buscó implementar una arquitectura que pudiera dividir la aplicación en tres componentes autónomos, cada uno cumpliendo un rol específico:
    - **Frontend:** Encargado de la interfaz de usuario y la interacción directa con los usuarios.
    - **Backend:** Responsable de la lógica del negocio y el procesamiento de las solicitudes realizadas a través de la API REST.
    - **Base de datos:** Almacenamiento centralizado de las tareas y su estado.
      La integración eficiente entre estos componentes se asegura mediante una API REST que utiliza métodos estándar como GET, POST, PUT y DELETE.
- **Realizar el despliegue de la aplicación en tres entornos distintos:**
    - Local: se utilizó Docker Compose para poder contenerizar todos los componentes.
    - Semilocal: se desplegó el frontend y el backend localmente con una base de datos alojada en AWS RDS, pues se observó que tenía mucha facilidad para el uso de esta.
    - Nube: ejecutando la solución completa en la nube mediante Docker Compose.

Solución Local

Se utiliza un repositorio de GitHub diferente para cada solución, aquí está el de la solución local: https://github.com/JuanPE-PNG/ToDo-app

La solución local se implementa utilizando Docker Compose para contenerizar y gestionar los tres componentes principales de la aplicación que serían backend, base de datos y frontend. La arquitectura aprovecha una red interna de Docker para garantizar una comunicación eficiente y segura entre los contenedores.

Diseño y separación de componentes:

- **Base de Datos:**

  Imagen: postgres:latest.

  Configurada con variables de entorno que establecen el usuario (admin), contraseña (password) y el nombre de la base de datos (tododb).

  Utiliza un volumen que carga un archivo de inicialización (./database/init.sql) al momento de arrancar el contenedor. Este archivo contiene la definición de tablas y datos iniciales necesarios para el funcionamiento de la aplicación.

  Expone el puerto 5432 para comunicación interna entre servicios.

  Se utiliza el comando pg_isready que verifica la disponibilidad de la base de datos incluyendo un healthcheck.

  El archivo init.sql define las siguientes estructuras y datos iniciales:

  Tablas:

  users: Para almacenar información de los usuarios, como nombre de usuario, correo electrónico y contraseñas encriptadas.

  categories: Para clasificar las tareas en categorías predefinidas como Trabajo, Estudios, Personal, etc.

  tasks: Para registrar las tareas con campos como título, descripción, estado de completado, fechas de creación y relaciones con usuarios y categorías.

  tags: Para gestionar etiquetas que pueden asignarse a las tareas.

  task_tags: Una tabla intermedia para implementar una relación de muchos a muchos entre tareas y etiquetas.

  Datos iniciales:

  Se insertan usuarios predeterminados (john_doe y jane_smith) con correos electrónicos y contraseñas ficticias.

  Se definen categorías como Trabajo, Estudios y Personal para clasificar las tareas.

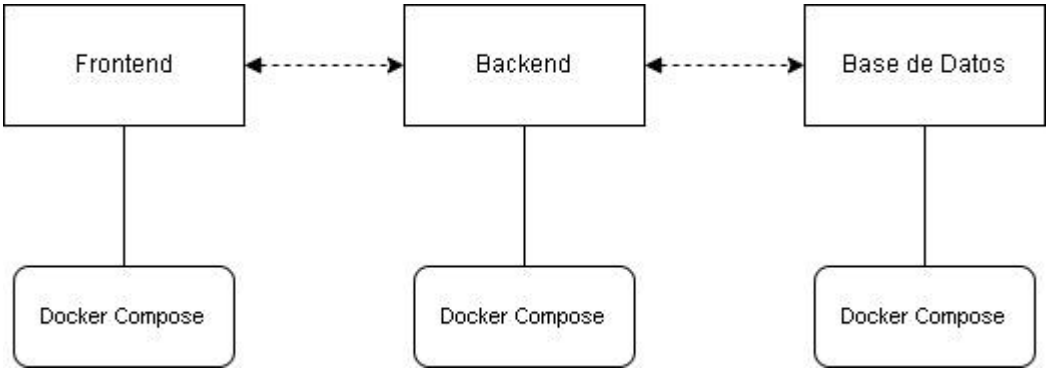
- **Backend:**

  El backend implementa la lógica de la aplicación y expone una API REST que interactúa con la base de datos PostgreSQL para manejar las operaciones CRUD.

En este componente encontramos el dockerfile para el backend, el cual se construye con la imagen, el directorio de trabajo, las dependencias, el archivo del código, el puerto expuesto y el comando inicial.

También está el archivo index.js, que es quien contiene el Middleware, el cual utiliza express.json() para analizar solicitudes JSON y cors para permitir solicitudes de diferentes orígenes.

Para la conexión PostgreSQL, esta fue configurada mediante request de los datos, como tal, la base de datos es la única que se comunica con el backend, donde se conecta y pide request a la api rest, todo esto mediante la configuración mediante variables de entorno (PG_USER, PG_PASSWORD, PG_DB, PG_HOST, PG_PORT).



- **Frontend:**

  El frontend es una aplicación desarrollada con React que interactúa con el backend a través de la API REST para realizar las operaciones CRUD. Se encarga de proporcionar una interfaz gráfica sencilla e intuitiva para la gestión de tareas.}

  El Dockerfile del frontend sigue los pasos estándar para construir y servir una aplicación React:

  Etapa 1: Construcción de la aplicación React

  Etapa 2: Servir la aplicación con Nginx

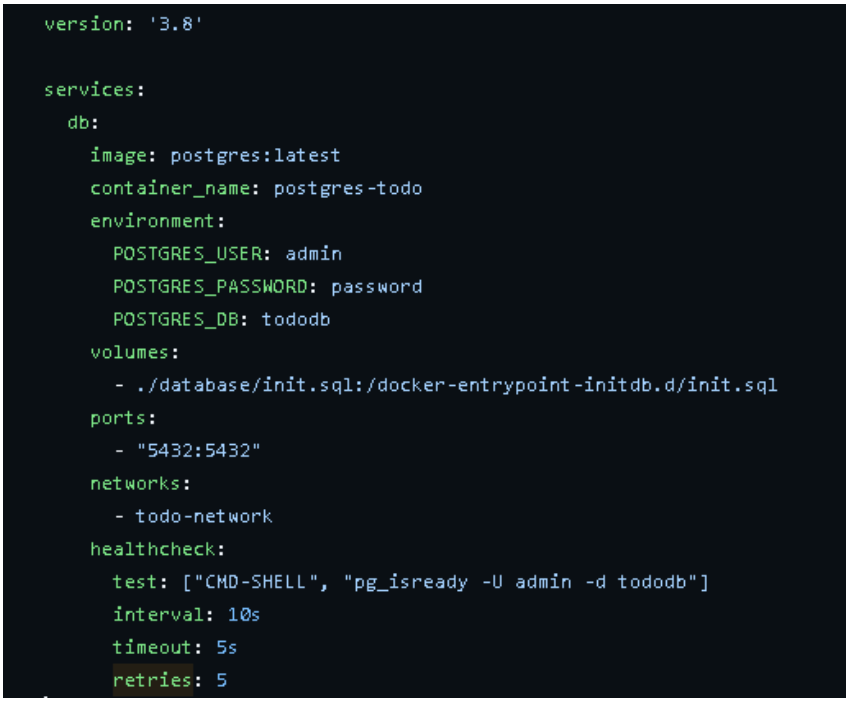
  Gestión de Tareas: El sistema de gestión de tareas proporciona una interfaz para visualizar, añadir, editar y eliminar tareas dentro de la aplicación. Estas funcionalidades permiten que los usuarios gestionen eficientemente sus actividades y tareas diarias.

  Interacción con el Backend: La comunicación entre el frontend y el backend es fundamental para que la aplicación funcione correctamente, ya que permite que el frontend realice operaciones de lectura y escritura en la base de datos.

Configuración de cada contenedor y cómo estos interactúan:

El proyecto utiliza Docker Compose para gestionar los contenedores que componen la aplicación. Estos contenedores están divididos en tres componentes principales: el backend, la base de datos y el frontend. Cada uno de estos contenedores se configura de manera independiente pero están conectados entre sí a través de una red definida en el archivo docker-compose.yaml.

Para la **Base de Datos** el contenedor de la base de datos se configura utilizando la imagen oficial de PostgreSQL. Su configuración incluye los parámetros necesarios para la conexión a la base de datos, como el nombre de usuario, la contraseña y el nombre de la base de datos. Además, se establece un volumen para la persistencia de datos y un script de inicialización (init.sql) que configura las tablas y relaciones en la base de datos al momento de iniciarse el contenedor.



```yaml
version: '3.8'

services:
  db:
    image: postgres:latest
    container_name: postgres-todo
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: password
      POSTGRES_DB: tododb
    volumes:
      - ./database/init.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "5432:5432"
    networks:
      - todo-network
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U admin -d tododb"]
      interval: 10s
      timeout: 5s
      retries: 5
```

Para la parte del **Backend** está compuesto por una aplicación Node.js que utiliza Express para manejar las solicitudes HTTP. Este contenedor se configura para conectarse al contenedor de la base de datos a través de la red definida en Docker Compose. La configuración también incluye las variables de entorno necesarias para establecer la

conexión con PostgreSQL.

```yaml
  backend:
    build:
      context: ./backend
    container_name: backend
    environment:
      - PG_USER=admin
      - PG_PASSWORD=password
      - PG_DB=tododb
      - PG_HOST=db
      - PG_PORT=5432
      - PORT=5000
    ports:
      - "5000:5000"
    depends_on:
      db:
        condition: service_healthy
    networks:
      - todo-network
    volumes:
      - ./backend:/app
      - /app/node_modules
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:5000 || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 5
```

Ya en el **Frontend** es una aplicación React que se ejecuta dentro de un contenedor configurado para servir la aplicación con Nginx. En la configuración de este contenedor, se especifica que el contenedor nginx debe servir los archivos estáticos generados en la etapa de construcción de la aplicación React.

```yaml
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: frontend
    ports:
      - "8080:80"
    environment:
      - REACT_APP_BACKEND_URL=http://backend:5000
    depends_on:
      backend:
        condition: service_healthy
    networks:
      - todo-network

networks:
  todo-network:
    driver: bridge
```

Al final, se muestra como todos los contenedores están conectados a una red interna de Docker llamada todo-network, que permite la comunicación entre ellos.

Solución en la Nube

Se realiza en el repositorio: https://github.com/JuanPE-PNG/ToDo-app-BDN

Replicación de la Arquitectura Local en la Nube (AWS y Google Cloud):

Se sube la base de datos utilizando servicios de AWS RDS para crear y manejar la base de datos en la nube. Se tiene una configuración pública orientada a cualquier red IPv4(0.0.0.0/0) para que cualquier persona que tenga la aplicación y desee utilizarla pueda acceder a la base de datos y administrar sus tareas. Se utiliza el servicio gratuito que ofrece AWS usando 10GB de espacio SSD para el correcto traspaso de información entre el backend y la base de datos.

Para el backend y el frontend de nuestra aplicación, se decidió utilizar Google Cloud para manejar nuestros servicios en la nube. Mediante los servicios Cloud Run, Cloud Build, Container Registry y por medio del CLI de Cloud, se subió, administró y configuró la aplicación para su correcto funcionamiento en la nube.

Se decidió por implementar la tecnología Docker para esta aplicación debido a su facilidad de uso y por la sencillez de esta. Por medio del archivo cloudbuild.yaml se construyen los contenedores y se suben las imágenes a la nube, donde mediante el comando en CLI (gcloud run deploy …) se pudo desplegar sencillamente la aplicación mediante las imágenes y contenedores en la nube.

Configuración del balanceador de carga para el backend replicado tres veces

Mediante los servicios de Google Cloud (Cloud Run) se logró replicar fácilmente el backend por medio de la configuración que obliga a mantener tres veces activo el servicio mediante la configuración en la sección de "Cantidad mínima de instancias del servicio".

Uso de servicios en la nube, configuración de redes y seguridad

Servicios en la Nube Utilizados:

- **Base de Datos:** Se utiliza AWS RDS para alojar la base de datos PostgreSQL, con alta disponibilidad y opciones de escalabilidad automática. Configurado en la zona de disponibilidad us-east-1a, una VPC estándar con 6 subredes y una tabla de enrutamiento. La base de datos es accesible públicamente mediante el uso y manejo de la aplicación.

- **Backend y Frontend**: Contenedores Docker desplegados en un servicio de contenedores gestionados mediante Container Registry de Google Cloud. Ahí es

donde se almacenan las imágenes y contenedores de nuestra aplicación además de permitir la fácil modificación y/o conectividad con nuestros repositorios.

Descripción del flujo para subir la aplicación a la nube

Primer paso:
Se verifican los Dockerfiles creados para el backend y el frontend ya que estos van a generar las imagenes y los contenedores que utilizaremos en la nube.

No se utiliza ahora el docker-compose si no que utilizaremos el propio orquestador que maneja Google Cloud.

Segundo paso:
Se construye el archivo cloudbuild.yaml para la construcción del contenedor y las imágenes del proyecto, estas se orquestan y manejan mediante Container Registry. A continuación la configuración general:

```
steps:
  # Construir el contenedor del backend
  - name: 'gcr.io/cloud-builders/docker'
    args: ['build', '-t', 'gcr.io/[PROJECT_ID]/backend', './backend']

  # Construir el contenedor del frontend
  - name: 'gcr.io/cloud-builders/docker'
    args: ['build', '-t', 'gcr.io/[PROJECT_ID]/frontend', './frontend']

  # Subir las imágenes al Container Registry
  - name: 'gcr.io/cloud-builders/docker'
    args: ['push', 'gcr.io/[PROJECT_ID]/backend']

  - name: 'gcr.io/cloud-builders/docker'
    args: ['push', 'gcr.io/[PROJECT_ID]/frontend']

images:
  - 'gcr.io/[PROJECT_ID]/backend'
  - 'gcr.io/[PROJECT_ID]/frontend'
```

Donde PROJECT_ID es la identificación proveída por la creación del proyecto en la plataforma.

Se configura y sube mediante el siguiente comando:
gcloud builds submit --config cloudbuild.yaml

Esto creará y subirá las imágenes a la API.

Tercer paso:

Cada que se desee actualizar algún contenedor o imagen, se usará **gcloud builds submit --tag …** .

Un ejemplo en nuestra aplicación fue:
gcloud builds submit --tag gcr.io/todo-app-4457024/frontend ./frontend

Cuarto paso:
Para subir a la nube y finalmente utilizar nuestra aplicación, se utiliza un comando el cual permite manejar la imagen, la plataforma y los permisos, además de la región en la que se va a desplegar.

En nuestra aplicación se utilizó, por ejemplo:

```
gcloud run deploy frontend \
  --image gcr.io/todo-app-4457024/frontend \
  --platform managed \
  --allow-unauthenticated \
  --region us-central1
```

Análisis y Conclusiones:

Rendimiento de la Aplicación: Local vs. Nube

En términos de rendimiento, la aplicación mostró no diferencias significativas cuando se comparó su ejecución en el entorno local y en la nube, sin embargo si se muestra un cambio según el entorno.

Ambiente Local: Al ejecutar la aplicación en el entorno local utilizando Docker Compose, la latencia era generalmente baja y los tiempos de respuesta de la base de datos eran rápidos, ya que todo el sistema (backend, frontend y base de datos) se ejecutaba en la misma máquina. Sin embargo, todo se ve limitado al equipo en donde se trabaje.

Ambiente en la Nube: Al desplegar la aplicación en la nube, el rendimiento mejoró gracias a la infraestructura optimizada de la nube, especialmente con el uso de instancias dedicadas para la base de datos y los servicios de backend. La nube también permitió una mejor distribución de carga a través de balanceadores de carga, lo que resultó en tiempos de respuesta más estables y rápidos bajo mayores cargas. La latencia en la nube fue algo mayor debido a la comunicación a través de redes externas, pero no hubo un impacto notable en el rendimiento.

Latencia, Escalabilidad y Disponibilidad

Latencia: La latencia en el entorno local era menor, ya que todos los servicios estaban en una misma red local. En la nube, la latencia aumentó ligeramente debido a la comunicación entre servicios distribuidos en diferentes redes (por ejemplo, el backend alojado en contenedores Docker y la base de datos en AWS RDS). Sin embargo, este incremento de

latencia fue mínimo y apenas perceptible para los usuarios finales, gracias a la optimización de las redes en la nube.

Escalabilidad: El entorno local limitó la escalabilidad de la aplicación, ya que no se podía aumentar fácilmente el número de instancias del backend o el acceso a la base de datos sin realizar una reconfiguración significativa del sistema local. Sin embargo, en la nube, la escalabilidad se gestionó mucho más fácilmente utilizando servicios de autoescalado para el backend y la base de datos, permitiendo agregar más instancias según la demanda.

Disponibilidad: La disponibilidad en el entorno local dependía de los recursos físicos de la máquina y su configuración, lo que podría afectar la continuidad del servicio en caso de fallos. En la nube, el sistema se benefició de una alta disponibilidad gracias a la infraestructura distribuida, respaldos automáticos de la base de datos y el uso de múltiples zonas de disponibilidad en AWS. Esto hizo que la aplicación fuera más resiliente ante fallos y permitiera una recuperación más rápida.

Retos y Cómo Fueron Abordados

Durante el despliegue de la aplicación, no se enfrentaron tantos retos como uno creería, sin embargo hubo uno bastante presente::

Problemas de Conexión entre Contenedores: En el entorno local, la configuración de redes en Docker Compose resultó en algunos problemas de comunicación entre los contenedores.Esto se pudo resolver ajustando la configuración de redes y utilizando variables de entorno para garantizar que los contenedores pudieran comunicarse de forma segura.

Reflexiones sobre el Uso de Tecnologías como Docker y Kubernetes

Realmente, nosotros como equipo de trabajo vimos que trabajar con Docker fue mucho más fácil y entretenido en su totalidad, por eso, el proyecto tiene ese enfoque, pues permitió crear un entorno aislado y reproducible para los contenedores de backend y frontend, asegurando que las aplicaciones se ejecutarán de manera consistente tanto en el entorno local como en la nube.

Mostrando así, como docker brinda una forma flexible y eficiente de manejar la infraestructura y mejorar la portabilidad de la aplicación.

Elementos Clave al Analizar un Proyecto de esta índole

Escalabilidad: El sistema se plantea para que sea capaz de manejar picos de tráfico y la adición de más usuarios sin comprometer el rendimiento.

La fiabilidad: es esencial para mantener el sistema operativo y funcional bajo condiciones diversas, como fallos de hardware o picos inesperados de tráfico. El uso de balanceadores de carga para autoescalado, y el uso de AWS RDS para garantizar la alta disponibilidad de la base de datos aumentan la fiabilidad del sistema.

Costo: En cuanto al costo, la implementación en la nube permitió una gestión flexible de los recursos, con la ventaja de un modelo de pago por uso. Durante el período de pruebas y despliegue de la aplicación en Google Cloud, se incurrió en un gasto total de $1.50, lo que refleja la eficiencia de la infraestructura en la nube para una aplicación pequeña en su fase inicial.

Optimización de Recursos: En la nube, los recursos deben ser gestionados con cuidado para evitar sobrecostos. El uso de contenedores y servicios gestionados ayuda a optimizar el uso de recursos virtuales, pero también es fundamental ajustar la capacidad de procesamiento y el almacenamiento según las necesidades reales del sistema. Pero en sí, todo depende directamente de la complejidad del proyecto/aplicación, en nuestro caso, la ToDoList es bastante sencilla, por lo que manejó un costo de $1.50 dólares.

Seguridad: En cuanto a la seguridad en la parte local:En el entorno local, esta fue asegurada mediante la configuración adecuada de los archivos Dockerfile y la utilización de redes internas aisladas, limitando el acceso a los contenedores y los servicios solo a aquellos que necesitan comunicarse entre sí.

Ya en la nube, se intentó reforzar la seguridad utilizando mecanismos de autenticación y autorización, así como cifrado de datos en tránsito y en reposo. Las redes virtuales fueron configuradas para aislar el tráfico de la aplicación y proteger los datos contra accesos no autorizados.

Además, al manejar las variables de entorno asegura que solo el backend acceda a ellas.

Conclusión Final

La migración de la solución a la nube trajo consigo varias mejoras en términos de escalabilidad, disponibilidad y rendimiento. Sin embargo, el proceso también presentó desafíos, especialmente en la configuración de la infraestructura en la nube. El uso de tecnologías Docker permitió que la aplicación fuera más flexible y portátil, facilitando tanto el despliegue en diferentes entornos como la gestión de la infraestructura. La optimización de la seguridad y la gestión de costos será fundamental para la continuidad y el éxito del sistema a largo plazo.