INFORME

(Grupo F)

(Proyecto)

Estudiantes Grupo 50:

Diego Alejandro Gomez Puentes – 202411026

Diego Alejandro Tolosa Sánchez – 202313023

Estudiantes Grupo 51:

Daniel Meléndez Ramirez – 202313024

Joan Sebastián Saavedra Perafán – 202313025

Asignatura:

Infraestructuras Paralelas y Distribuidas

Docente:

Carlos Andres Delgado Saavedra

Ingeniería de Sistemas

Universidad del Valle – Sede Tuluá

25 de diciembre de 2024

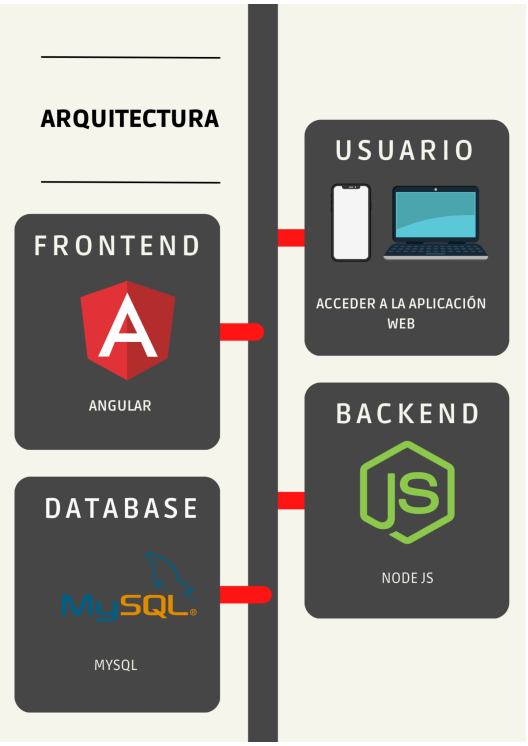
Tabla de Contenidos

Introducción

Esta aplicación web logra automatizar el proceso de comercialización y el control administrativo de una empresa llamada “Café Rosita” permitiendo gestionar el registro de ventas y procesos de manufacturación, obteniendo la información requerida de manera más eficiente, además de evitar problemas a futuro con el respaldo de la información.

Las tecnologías utilizadas por el equipo de desarrollo para crear esta aplicación son:

- Angular en su versión **16.0.0** para el Front-End
- NodeJS en su versión **22.11.0** para el Back-End
- MySQL/MariaDB en su versión **3.3.2** como gestor de Base de datos



Durante el proceso de desarrollo se trabajó de manera integral con la empresa que cuenta con diversas secciones o áreas de trabajo que fueron representadas en la aplicación bajo los módulos de: gestionar clientes, empleados, pedidos, productos, proveedores, ventas, manufacturación, gastos, compras, reportes, entre otros; y a su vez los roles definidos para administrar los módulos de la aplicación basados en la estructura de la empresa son: administrador, empleado y cliente.

Objetivos

Objetivo General

Desarrollar un aplicativo web que permita gestionar el registro de ventas, procesos de manufacturación, generar reportes necesarios para mejorar la calidad en el servicio y la comercialización de los productos mediante la creación de una tienda virtual.

Objetivos Específicos

- Realizar un análisis de los requerimientos para establecer los alcances y limitaciones del sistema
- Diseñar el sistema y estructurar la base de datos
- Realizar pruebas funcionales y de diseño, con el fin de asegurar una buena aceptación por parte de los usuarios
- Realizar análisis con el usuario por medio de entrevistas, para poder solucionar detalles de la implementación del sistema
- Investigar tecnologías y lenguajes de programación para implementarlos en nuestro sistema
- Reuniones con el cliente para indagar problemas de usabilidad encontrados en los prototipos
- Diseñar prototipos acordes a las necesidades que nos planteó la clienta

Solución local

En este curso se dockeriza la aplicación antes descrita para brindar más seguridad, escalabilidad, flexibilidad, mantenibilidad, una planificación más ágil por parte de la gestión empresarial, mejor comunicación internamente, manteniendo la gestión de procesos en tiempo real y accesibilidad desde cualquier lugar.

De acuerdo al diagrama de separación de componentes el usuario sólo se comunica con la aplicación a través del Front-End, mientras que la API REST administra la comunicación desde y hacia el Back-End, donde este último hace las operaciones CRUD a la Base de datos que a su vez, comunica los resultados de las consultas al Back-End.

Docker Compose

Dockerfile del Front-End

Cabe señalar que durante las pruebas con Docker la imagen node:21 nuestro servidor Node.js tuvo mal rendimiento, conllevando un cambio hacia la rama node:21-alpine que es más ligera, pero generaba conflicto y no funcionaba la aplicación, por lo que se opta por la versión node:18 para facilitar la ejecución de aplicaciones Node en el contenedor.

Este archivo Dockerfile configura y construye una imagen Docker para una aplicación cuyo Front-end fue creado en Angular, gestionando dependencias para que las librerías estén disponibles, configurando el entorno de trabajo con las rutas de archivos designadas y asegurando que la pueda ser accedida correctamente. Lo anterior, con el fin de facilitar el la construcción y despliegue del contenedor, junto a la ejecución de la aplicación en cualquier entorno que soporte Docker.

Una vez se encuentra en ejecución la imagen, se crea un volumen en Docker para persistir los datos de configuración o archivos generados y utilizados por el contenedor. Permitiendo que los datos se mantengan fuera del ciclo de vida del contenedor, asegurando que no se pierdan cuando el contenedor se reinicie o elimine, sobre todo durante el desarrollo y con todas las pruebas realizadas para asegurar su correcto funcionamiento.

Al ejecutar todo correctamente, podremos pasar nuestro contenedor a un entorno consistente de producción, y ser accedido desde diferentes navegadores web.

```
# Se construye la imagen utilizamos una versión de Node v18 y referenciada como builder
FROM node:18 AS builder

# Se establece el directorio de trabajo
WORKDIR /app

# Copiamos archivos esenciales de configuración e instalación de dependencias
COPY package*.json ./
RUN npm install

# Copiamos el código fuente  y se compila la aplicación en Angular
COPY . .
RUN npm run build –prod

# Se sirve la aplicación con una edición Alpine de NGinx
FROM nginx:alpine

# Copiamos los archivos compilados desde la etapa de construcción
COPY –from=builder /app/dist/app-frontend /usr/share/nginx/html

# Copiamos el archivo default.conf a la ubicación correcta de Nginx
COPY default.conf /etc/nginx/conf.d/default.conf

# Exponemos el puerto que escuchará el contenedor para la aplicación en Angular
EXPOSE 4200

# Comando para iniciar la aplicación y hacerlo accesible desde fuera del contenedor
CMD [“nginx”, “ -g”, “daemon off;”]
```

Dockerfile del Back-End

En este caso utilizamos la imagen node:18 debido a problemas de compatibilidad entre dependencias que ejecutamos en la aplicación con la imagen node:21 al igual que el Front-End, para facilitar la ejecución de aplicaciones Node en el contenedor.

Este archivo Dockerfile configura y construye una imagen Docker para una aplicación cuyo Back-End fue creado en Node.js, gestionando dependencias para que las librerías estén disponibles, configurando las variables del entorno de trabajo con las rutas de archivos designadas y asegurando que la aplicación pueda ser accedida correctamente, una vez la base de datos esté disponible, para ejecutar la lógica del servidor y manejar solicitudes o procesamiento de datos, en cualquier entorno que soporte Docker.

Como sabemos, la imagen es una plantilla estática porque tiene una estructura base predeterminada, que una vez en ejecución se crea un volumen en Docker para almacenar los datos de configuración o archivos generados y utilizados por el contenedor. Permitiendo que se mantengan fuera del ciclo de vida del contenedor, asegurando que no se pierdan cuando se reinicie o elimine, sobre todo durante el desarrollo con todas las pruebas realizadas para asegurar su correcto funcionamiento.

Al ejecutar todo correctamente, podremos pasar nuestro contenedor a un entorno consistente de producción, con todos los servicios necesarios funcionando y ser accedido a través del Front-End recibiendo peticiones de los usuarios o también dicho contenedor puede interactuar con la aplicación como una base de datos.

```
# Se construye la imagen utilizamos una versión de Node v18
FROM node:18

# Establecemos variable de entorno para optimizar la aplicación en entorno de producción
ENV NODE_ENV=production

# Creamos y establecemos el directorio de trabajo dentro del contenedor
WORKDIR /usr/src/app

# Copiamos el archivo package.json y package-lock.json (si existe) e instala dependencias
COPY package*.json ./
RUN npm install

# Instalamos sequelize globalmente
RUN npm install -g sequelize

# Actualizamos paquetes e instalamos netcat-openbsd (nc) para el script wait-for-it.sh
RUN apt-get update && apt-get install -y netcat-openbsd

# Copiamos el contenido del directorio de trabajo y el código fuente de la aplicación
COPY . .

# Copiamos el script wait-for-it.sh y le otorgamos permisos de ejecución
COPY wait-for-it.sh /usr/local/bin/wait-for-it.sh
RUN chmod +x /usr/local/bin/wait-for-it.sh

# Exponemos el puerto que escuchará el contenedor para la aplicación
EXPOSE 3000

# Comando ejecutar script de espera wait-for-it.sh hasta que el servicio db esté disponible
# Para luego ejecutar el comando necesario para iniciar la aplicación
CMD /usr/local/bin/wait-for-it.sh db -- npm start
```

Script del Back-End

Utilizamos el script wait-for-it.sh para verificar que el servicio MySQL esté disponible antes de continuar con la configuración del Back-End, aplicando los siguientes comandos, configurando a su vez el entorno preparando la base de datos al aplicar las migraciones y seeders para luego iniciar la aplicación.

```
#!/bin/bash

# Aseguramos que el script se detendrá si cualquier comando falla en su ejecución
set -e

host="$1"  #Este argumento corresponde al nombre o dirección del host de base de datos
shift  #Eliminamos el primer argumento (host) de la lista de parámetros
cmd="$@"  #Capturamos los argumentos restantes que se desean ejecutar al final

until nc -z "$host" 3306; do  #Verifica con nc (Netcat)si el puerto 3306 del host está abierto
  echo "Esperando a que MySQL esté listo en $host:3306..."  #Mensaje indicando la espera
  sleep 2  #Se esperan 2 segundos antes de volver a intentarlo
done  #El bloque until ejecuta un bucle hasta que nc detecta que el puerto está accesible

# Nos aseguramos de ejecutar migraciones de las tablas hacia la base de datos usando npx
echo "Ejecutando migraciones..."
npx sequelize db:migrate

# Nos aseguramos de ejecutar seeders o registros hacia la base de datos usando npx
echo "Ejecutando seeders..."
npx sequelize db:seed:all

echo "Iniciando la aplicación..."
exec $cmd  #Reemplaza el proceso actual liberando recursos y ejecuta el proceso principal
```

Base de datos

A diferencia del Front-End y del Back-End, la base de datos no cuenta con Dockerfile, ya que utilizamos un extracto del archivo docker-compose.yml, específicamente un servicio llamado db.

Este extracto configura la base de datos MySQL como un servicio utilizando una imágen preconstruida disponible en Docker Hub que incluye el servidor de la base de datos, agregando las opciones necesarias para construcción, acceso, persistencia de datos y configuración de red, lo que facilita la implementación y gestión de la base de datos en un entorno Dockerizado.

Para nuestra base de datos, utilizamos el volumen para almacenar archivos de la base de datos, como por ejemplo: las imágenes de la empresa, productos y clientes, entre otros. Todo dentro de un contenedor, permitiendo la interacción con la base de datos a través de consultas, inserciones, actualizaciones, entre otros que solicite en este caso, el contenedor del Back-End para operaciones de lectura/escritura.

```
db:  #Definimos un servicio llamado database
  image: mysql:9.1.0  #Utilizamos una imagen oficial mysql en su versión 9.1.0
  environment:  #Se configuran los valores de la base de datos y del usuario
    MYSQL_ROOT_PASSWORD: root  #Contraseña del usuario root en la base de datos
    MYSQL_DATABASE: caferosita  #Nombre de la base de datos del proyecto
  ports:  #Configuramos los puertos
    - "3306:3306"  #Exponemos el puerto 3306 del contenedor al puerto 3306 del host
  volumes:  #Utilizamos un volumen llamado db_data
    - db_data:/var/lib/mysql  #Creamos y montamos un volumen dentro del contenedor
  networks:  #Referenciamos las redes
    - my-network  #Conectamos el servicio para que otros servicios se comuniquen con db
```

Docker-compose con imágenes locales

En este archivo docker-compose.yml configuramos tres servicios que corresponden al Front-End, Back-End y Base de datos, cada uno con sus propias especificaciones de red, puertos, y variables de entorno. También definimos las redes necesarias para la comunicación entre estos servicios.

```
services:
  frontend:  #Definimos un servicio llamado frontend
    image: angular-dev  #Nombramos la imagen para construir el contenedor
    container_name: frontend  #Asignamos un nombre al contenedor
    ports:
      - "4200:4200"  #Exponemos el puerto 4200 del contenedor al puerto 4200 del host
    environment:
      - NODE_ENV=development  #Variable de entorno para ejecutar en producción
    depends_on:
      - backend  #Aseguramos que el backend inicie antes, ya que depende de él
    networks:
      - my-network  #Conecta el servicio para que otros servicios se comuniquen con el front

  backend:  #Definimos un servicio llamado backend
    image: my-backend  #Nombramos la imagen para construir el contenedor
    container_name: backend  #Asignamos un nombre al contenedor
    ports:
      - "3000:3000"  #Exponemos el puerto 3000 del contenedor al puerto 3000 del host
```

```
  environment:
    - NODE_ENV=development  #Variable de entorno para ejecutar en producción
    - PORT=3000  #Establecemos el puerto de comunicación de la aplicación
  volumes:
    - ./backend:/app/backend  #Monta el directorio dentro del contenedor
  networks:
    - my-network  #Conecta el servicio para que otros servicios se comuniquen con el back
  depends_on:
    - db  #Definimos que el backend depende de que la base de datos esté disponible

db:  #Definimos un servicio llamado database
  image: mysql:9.1.0  #Utilizamos una imagen oficial mysql en su versión 9.1.0
  environment:  #Se configuran los valores de la base de datos y del usuario
    MYSQL_ROOT_PASSWORD: root  #Contraseña del usuario root en la base de datos
    MYSQL_DATABASE: caferosita  #Nombre de la base de datos del proyecto
  ports:  #Configuramos los puertos
    - "3306:3306"  #Exponemos el puerto 3306 del contenedor al puerto 3306 del host
  volumes:  #Utilizamos un volumen llamado db_data
    - db_data:/var/lib/mysql  #Creamos y montamos un volumen dentro del contenedor
  networks:  #Referenciamos las redes
    - my-network  #Conectamos el servicio para que otros servicios se comuniquen con db

networks:
  my-network:  #Definimos una red llamada my-network
    driver: bridge  #Utilizamos bridge en la red para comunicar los contenedores

volumes:
  db_data:  #Declaramos el volumen
```

Docker-compose con repositorio del docker hub

https://hub.docker.com/repository/docker/sebastian2313025/infraestructura-saavedra/general

Este archivo docker-compose.yml configura tres servicios (frontend, backend, y database), cada uno con sus propias especificaciones de red, puertos, y variables de entorno. También define las redes necesarias para la comunicación entre estos servicios.En este se usa la configuración del repositorio del docker hub trayendo las imágenes.

```
services:
  frontend:  #Definimos un servicio llamado frontend
    image: sebastian2313025/infraestructura-saavedra:frontend  #Imagen en Docker Hub
    container_name: frontend  #Asignamos un nombre al contenedor
    ports:
```

```
    - "4200:4200"  #Exponemos el puerto 4200 del contenedor al puerto 4200 del host
   environment:
    - NODE_ENV=development  #Variable de entorno para ejecutar en producción
   depends_on:
    - backend  #Aseguramos que el backend inicie antes, ya que depende de él
   networks:
    - my-network  #Conecta el servicio para que otros servicios se comuniquen con el front

backend:  #Definimos un servicio llamado backend
  image: sebastian2313025/infraestructura-saavedra:backend  #Imagen en Docker Hub
  container_name: backend  #Asignamos un nombre al contenedor
  ports:
    - "3000:3000"  #Exponemos el puerto 3000 del contenedor al puerto 3000 del host
  environment:
    - NODE_ENV=development  #Variable de entorno para ejecutar en producción
    - PORT=3000  #Establecemos el puerto de comunicación de la aplicación
 volumes:
    - ./backend:/app/backend  #Monta el directorio dentro del contenedor
  networks:
    - my-network  #Conecta el servicio para que otros servicios se comuniquen con el back
  depends_on:
    - db  #Definimos que el backend depende de que la base de datos esté disponible

db:  #Definimos un servicio llamado database
  image: mysql:9.1.0  #Utilizamos una imagen oficial mysql en su versión 9.1.0
  environment:  #Se configuran los valores de la base de datos y del usuario
    MYSQL_ROOT_PASSWORD: root  #Contraseña del usuario root en la base de datos
    MYSQL_DATABASE: caferosita  #Nombre de la base de datos del proyecto
  ports:  #Configuramos los puertos
    - "3306:3306"  #Exponemos el puerto 3306 del contenedor al puerto 3306 del host
  volumes:  #Utilizamos un volumen llamado db_data
    - db_data:/var/lib/mysql  #Creamos y montamos un volumen dentro del contenedor
  networks:  #Referenciamos las redes
    - my-network  #Conectamos el servicio para que otros servicios se comuniquen con db

networks:
  my-network:  #Definimos una red llamada my-network
    driver: bridge  #Utilizamos bridge en la red para comunicar los contenedores

volumes:
  db_data:  #Declaramos el volumen
```

Kubernetes

Frontend-service.yaml

Definimos un servicio en Kubernetes que expone una aplicación Angular a través de un balanceador de carga, asegurando que el tráfico en el puerto 4200 del host se dirija al puerto 4200 en los pods asociados. Las anotaciones y etiquetas aquí definidas, proporcionan contexto adicional y facilitan la gestión del servicio dentro del clúster.

```
apiVersion: v1  #Versión de la API de Kubernetes que se está utilizando
kind: Service  #Indica el tipo de objeto como servicio y puede ejecutarse en varios pods

metadata:  #Proporcionamos datos al objeto de tipo servicio en este caso
  annotations:
    kompose.cmd: kompose convert -f docker-compose.yml  #Formato de docker a kubernet
    kompose.version: 1.35.0 (9532ceef3)  #Versión de kompose utilizada para la conversión
  labels:
    io.kompose.service: frontend  #Etiqueta para identificar este servicio con valor frontend
  name: frontend  #Establecemos el nombre del servicio como frontend

spec:  #Definimos las especificaciones del servicio
  type: LoadBalancer  #Se expondrá a través de un balanceador de carga externo
  ports:
    - name: "4200"  #Indicamos el nombre puerto
      port: 4200  #Puerto en el que el servicio está disponible
      targetPort: 4200  #Puerto al que se redirige el tráfico dentro de los pods
  selector:
    io.kompose.service: frontend  #Selector de etiquetas para identificar pods del frontend
```

Frontend-deployment.yaml

Se define un despliegue o deployment en Kubernetes para el Front-End de nuestra aplicación en Angular, donde especificamos que se debe ejecutar una réplica del pod frontend, utilizando la imagen Docker *sebastian2313025/infraestructura-saavedra:frontend*, con configuraciones de memoria y CPU ajustadas. El puerto 4200 se expone para acceder a la aplicación, y se establecen variables de entorno para el entorno de desarrollo.

```
apiVersion: apps/v1  #Versión de la API de Kubernetes que se está utilizando
kind: Deployment  #Indica que la aplicación va a desplegarse en uno o varios pods

metadata:  #Proporcionamos datos al objeto de tipo servicio en este caso
```

```yaml
  annotations:
    kompose.cmd: kompose convert -f docker-compose.yml  #Formato de docker-kubernete
    kompose.version: 1.35.0 (9532ceef3)  #Versión de kompose utilizada para la conversión
  labels:
    io.kompose.service: frontend  #Etiqueta para identificar este servicio con valor frontend
  name: frontend  #Establecemos el nombre del servicio como frontend

spec:  #Definimos las especificaciones del despliegue
  replicas: 1  #Se establecen la cantidad de réplicas del pod
  selector:  #Definimos como seleccionar los pods al momento del despliegue
    matchLabels:
      io.kompose.service: frontend  #Seleccionamos los pods que tienen la etiqueta frontend
  template:
    metadata:
      annotations:
        kompose.cmd: kompose convert -f docker-compose.yml  #Formato docker-kubernetes
        kompose.version: 1.35.0 (9532ceef3)  #Versión de kompose utilizada para conversión
      labels:
        io.kompose.service: frontend  #Establecemos las etiquetas para los pods

    spec:  #Definimos las especificaciones para los contenedores dentro del pod
      containers:  #Lista de contenedores que se ejecutarán en el pod
        - env:
            - name: NODE_ENV  #Variable de entorno para ejecutar en producción
              value: development  #Valor que indica que va ser desplegado
          image: danyel3096/infra-para-dist:angular-dev  #Imagen del docker que utilizamos
          name: frontend  #Nombre del contenedor

          ports:
            - containerPort: 4200  #El puerto asignado al contenedor es 4200
          resources:
            limits:
              memory: "128Mi"  #Asigna 128Mi de memoria al contenedor
              cpu: "100m"  #Asigna 100m como límite de CPU
            requests:  #Definimos las solicitudes de recursos a garantizar para funcionar
              memory: "128Mi"  #Se garantiza 128Mi de memoria
              cpu: "100m"  #Se garantiza 100m de CPU
      restartPolicy: Always  #Política que garantiza el auto-reinicio del pod, si falla
```

Backend-service.yaml

Definimos un servicio en Kubernetes que expone una aplicación backend a través del puerto 3000. Las anotaciones y etiquetas que estructuramos proporcionan contexto adicional y facilitan la gestión del servicio dentro del clúster. El servicio Back-End utiliza un selector para dirigir el tráfico a los pods que tienen la etiqueta *io.kompose.service: backend*.

```yaml
apiVersion: v1   #Versión de la API de Kubernetes que se está utilizando
kind: Service   #Indica el tipo de objeto como servicio y puede ejecutarse en varios pods

metadata:   #Proporcionamos datos al objeto de tipo servicio en este caso
  annotations:
    kompose.cmd: kompose convert -f docker-compose.yml #Formato de docker a kubernet
    kompose.version: 1.35.0 (9532ceef3)   #Versión de kompose utilizada para la conversión
  labels:
    io.kompose.service: backend   #Etiqueta para identificar este servicio con valor backend
  name: backend   #Establecemos el nombre del servicio como backend

spec:   #Definimos las especificaciones del servicio
  type: LoadBalancer  #Especificamos el tipo como Balanceador de carga
    ports:
      - name: "3000"   #Nombre de puerto
        port: 3000   #Puerto en el que el servicio está disponible
        targetPort: 3000   #Puerto al que se redirige el tráfico dentro de los pods
    selector:
      io.kompose.service: backend #Selector de etiquetas para identificar pods del backend
```

Backend-deployment.yaml

Se define un despliegue en Kubernetes para el Back-End de nuestra aplicación en NodeJs, donde se debe ejecutar una réplica del pod backend, utilizando la imagen Docker *danyel3096/infra-para-dist:back-dev*, con configuraciones de memoria y CPU ajustadas. El puerto 3000 se expone para acceder a la aplicación, y se establecen variables de entorno para el entorno de desarrollo. También se especifica un reclamo de volumen persistente para mantener los datos del contenedor.

```yaml
apiVersion: apps/v1   #Versión de la API de Kubernetes que se está utilizando
kind: Deployment   #Indica que la aplicación va a desplegarse en uno o varios pods

metadata:   #Proporcionamos datos al objeto para despliegue en este caso
  annotations:
    kompose.cmd: kompose convert -f docker-compose.yml #Formato de docker a kubernet
    kompose.version: 1.35.0 (9532ceef3) #Versión de kompose utilizada para la conversión
  labels:
    io.kompose.service: backend   #Etiqueta para identificar este servicio con valor backend
  name: backend   #Establecemos el nombre del servicio como backend

spec:  #Definimos las especificaciones del despliegue
  replicas: 3  #Establecemos el número de réplicas del pod
  selector:  #Definimos como queremos seleccionar los pods utilizando una etiqueta
    matchLabels:  #Seleccionamos los pods que coincidan con la etiqueta
      io.kompose.service: backend  #Asignamos la etiqueta como backend
```

```yaml
strategy:
  type: Recreate   #Indicamos que se terminan pods existentes antes de crear unos nuevos
template:    #Definimos la plantilla del pod que se creará
  metadata:   #Establecemos las etiquetas y anotaciones para los pods
    annotations:   #Proporcionamos contexto adicional sobre las conversión
      kompose.cmd: kompose convert -f docker-compose.yml
      kompose.version: 1.35.0 (9532ceef3)
    labels: #Proporcionamos las etiquetas para los pods
      io.kompose.service: backend   #Determinamos la etiqueta como backend

  spec:   #Definimos las especificaciones para los contenedores dentro del pod
    containers:   #Lista de contenedores que se ejecutará en el pod
      - env:   #Variables de entorno para configurar el contenedor
          - name: NODE_ENV   #Establecemos las variables del entorno para el contenedor
            value: development   #Definimos development como valor
          - name: PORT   #Definimos que puerto vamos a utilizar
            value: "3000"   #Asignamos el puerto 3000 para comunicar el contenedor
        image: danyel3096/infra-para-dist:back-dev
        name: backend   #Nombre del contenedor
        ports:   #Definimos el puerto que expondrá el contenedor
          - containerPort: 3000   #Asignamos el puerto 3000
            protocol: TCP   #Asignamos el protocolo TCP
        resources:   #Especificamos los recursos asignados al contenedor
          limits:   #Definimos los límites máximos de cada recurso
            memory: "128Mi"   #Limitamos la memoria a 128Mi
            cpu: "100m"   #Límite de CPU a 100m
          requests:  #Definimos las solicitudes de recursos garantizadas
            memory: "128Mi"  #Garantizamos 128Mi de memoria
            cpu: "100m"  #Garantizamos 100 de CPU
    restartPolicy: Always  #Política que garantiza el auto-reinicio del pod, si falla
```

Backend-claim0-persistentvolumeclaim.yaml

Este archivo corresponde al primer volumen que será asignado a la primera réplica que definimos antes en Kubernetes. Aquí, solicitamos 100 MiB de almacenamiento persistente con acceso de lectura y escritura para un solo nodo, es decir, la primera réplica. Además, las etiquetas y el nombre proporcionan un identificador único y facilitan la gestión del recurso dentro del clúster de Kubernetes.

```yaml
apiVersion: v1  #Especificamos la versión de la API de Kubernetes
kind: PersistentVolumeClaim  #Indicamos el tipo de objeto que se está definiendo

metadata:  #Proporcionamos los datos del objeto
  name: backend-claim0-pod0  #Establecemos el nombre del primer volumen
```

```
spec:  #Definimos las especificaciones
  accessModes:  #
    - ReadWriteOnce  #Permitimos que el volumen sea montado como lectura-escritura
  resources:  #
    requests:  #Definimos la cantidad de recursos limitados
      storage: 100Mi  #Solicitamos 100Mi de almacenamiento, reservados para el volumen
```

Backend-claim1-persistentvolumeclaim.yaml

Este archivo corresponde al segundo volumen que será asignado a la segunda réplica que definimos antes en Kubernetes. Aquí, solicitamos 100 MiB de almacenamiento persistente con acceso de lectura y escritura para un solo nodo, es decir, la segunda réplica. Además, las etiquetas y el nombre proporcionan un identificador único y facilitan la gestión del recurso dentro del clúster de Kubernetes.

```
apiVersion: v1  #Especificamos la versión de la API de Kubernetes
kind: PersistentVolumeClaim  #Indicamos el tipo de objeto que se está definiendo

metadata:  #Proporcionamos los datos del objeto
  name: backend-claim1-pod1  #Establecemos el nombre del segundo volumen

spec:  #Definimos las especificaciones
  accessModes:  #
    - ReadWriteOnce  #Permitimos que el volumen sea montado como lectura-escritura
  resources:  #
    requests:  #Definimos la cantidad de recursos limitados
      storage: 100Mi  #Solicitamos 100Mi de almacenamiento, reservados para el volumen
```

Backend-claim2-persistentvolumeclaim.yaml

Este archivo corresponde al primer volumen que será asignado a la tercera réplica que definimos antes en Kubernetes. Aquí, solicitamos 100 MiB de almacenamiento persistente con acceso de lectura y escritura para un solo nodo, es decir, la tercera réplica. Además, las etiquetas y el nombre proporcionan un identificador único y facilitan la gestión del recurso dentro del clúster de Kubernetes.

```
apiVersion: v1  #Especificamos la versión de la API de Kubernetes
kind: PersistentVolumeClaim  #Indicamos el tipo de objeto que se está definiendo

metadata:  #Proporcionamos los datos del objeto
  name: backend-claim2-pod2  #Establecemos el nombre del tercer volumen
```

```yaml
spec:  #Definimos las especificaciones
  accessModes:  #
    - ReadWriteOnce  #Permitimos que el volumen sea montado como lectura-escritura
  resources:  #
    requests:  #Definimos la cantidad de recursos limitados
      storage: 100Mi  #Solicitamos 100Mi de almacenamiento, reservados para el volumen
```

Db-service.yaml

Definimos un servicio en Kubernetes que expone nuestra base de datos MySQL a través del puerto 3306. A su vez, agregamos las anotaciones y etiquetas que proporcionan contexto adicional y facilitan la gestión del servicio dentro del clúster de Kubernetes. El servicio utiliza un selector para dirigir el tráfico a los pods que tienen la etiqueta *io.kompose.service: db*.

```yaml
apiVersion: v1  #Especificamos la versión de la API de Kubernetes
kind: Service  #Indicamos el tipo de objeto como servicio

metadata:  #Proporcionamos datos sobre el objeto
  annotations:  #
    kompose.cmd: kompose convert -f docker-compose.yml  #Convertimos con Kompose
    kompose.version: 1.35.0 (9532ceef3)  #Indicamos la versión para la conversión
  labels:  #
    io.kompose.service: db  #Etiqueta para identificar y seleccionar este servicio como db
  name: db  #Establecemos el nombre del servicio

spec:  #Definimos las especificaciones del servicio
  ports:  #Establecemos los puertos
    - name: "3306"  #Nombre del puerto, en este caso es el número del puerto
      port: 3306  #El puerto en el que el servicio está disponible
      targetPort: 3306  #El puerto al que se dirige el tráfico dentro de los pods
  selector:  #
    io.kompose.service: db  #Identificamos los pods que este servicio debe gestionar
```

Db-deployment.yaml

Se define un despliegue en Kubernetes para nuestra base de datos MySQL, que debe ejecutar una réplica del pod db, utilizando la imagen Docker mysql:9.1.0, con las configuraciones de volumen persistente para los datos. Finalmente, se expone el puerto 3306 para acceder a la base de datos, y se establecen variables de entorno para la configuración de MySQL.

```yaml
apiVersion: apps/v1   #Especificamos la versión de la API de Kubernetes
kind: Deployment   #Indicamos tipo de objeto para administrar una aplicación desplegada

metadata:   #Proporcionamos los datos sobre el objeto
  annotations:   #
    kompose.cmd: kompose convert -f docker-compose.yml   #Kompose para kubernetes
    kompose.version: 1.34.0 (HEAD)   #Indicamos la versión del kompose para conversión
  labels:   #
    io.kompose.service: db   #Etiqueta utilizada para identificar y seleccionar el servicio db
  name: db   #Establecemos el nombre del deployment

spec:   #Definimos las especificaciones del despliegue
  replicas: 1   #Establecemos que solo se ejecuta una sola réplica del pod
  selector:   #Seleccionamos los pods gestionados por este despliegue, con una etiqueta
    matchLabels:   #Seleccionamos los pods que tienen la sgte etiqueta con el valor db
      io.kompose.service: db   #
  strategy:   #
    type: Recreate   #Indica que los pods existentes se eliminan antes de crear unos nuevos
  template:   #Definimos la plantilla del pod que se crea
    metadata:   #Definimos las etiquetas y anotaciones para los pods
      annotations:   #Proporcionamos contexto adicional sobre la conversión
        kompose.cmd: kompose convert -f docker-compose.yml   #
        kompose.version: 1.34.0 (HEAD)   #
      labels:   #Proporcionamos etiquetas para los pods
        io.kompose.service: db   #
    spec:   #Definimos las especificaciones para los contenedres
      containers:   #Lista de contenedores que se ejeutan en el pod
        - env:   #Variables de entorno para configurar el contenedor
            - name: MYSQL_DATABASE   #Establecemos el nombre de la base de datos
              value: caferosita   #
            - name: MYSQL_ROOT_PASSWORD   #Establecemos la contraseña de root
              value: root   #
          image: mysql:9.1.0   #Imagen de Docker que se utiliza
          name: db   #Nombre del contenedor
          ports:   #Definimos el puerto que el contenedor expondrá
            - containerPort: 3306   #Establecemos el puerto en el contenedor
              protocol: TCP   #Protocolo utilizado
          resources:
            limits:
              memory: "512Mi"  #Asigna 512Mi de memoria al contenedor
              cpu: "500m"  #Asigna 500m como límite de CPU
            requests:  #Definimos las solicitudes de recursos a garantizar para funcionar
              memory: "256Mi"  #Se garantiza 256Mi de memoria
              cpu: "250m"  #Se garantiza 250m de CPU
          volumeMounts:   #Definimos los volúmenes que se montan en el contenedor
            - mountPath: /var/lib/mysql   #Ruta en el contenedor donde se monta el volumen
              name: db-data   #Nombre del volumen
      restartPolicy: Always   #Política para el auto-reinicio del pod en caso de que falle
```

```
    volumes:   #Definimos los volúmenes utilizados por el pod
     - name: db-data   #Nombre del volumen
       persistentVolumeClaim:   #Definimos el reclamos de volumen persistente
         claimName: db-data   #Nombre del reclamo
```

Db-data-persistentvolumeclaim.yaml

Definimos un servicio en Kubernetes, que solicita 100 MiB de almacenamiento persistente con acceso de lectura y escritura para un solo nodo correspondiente a nuestra base de datos. Sumado a lo anterior, tenemos las etiquetas y el nombre que proporcionan un identificador único y facilitan la gestión del recurso dentro del clúster de Kubernetes.

```
apiVersion: v1   #Versión de la API de Kubernetes que estamos utilizando
kind: PersistentVolumeClaim   #Indicamos tipo de objeto para almacenamiento persistente

metadata:   #Proporcionamos datos sobre el objeto
  labels:   #
    io.kompose.service: db-data   #Etiqueta utilizada para identificar el recurso
  name: db-data   #Establecemos el nombre del objeto como db-data

spec:   #Definimos las especificaciones del volumen
  accessModes:   #
    - ReadWriteOnce   #Montamos el volumen como lectura-escritura por un único nodo
  resources:   #
    requests:   #Definimos la cantidad de recursos solicitados
      storage: 100Mi   #Solicitamos 100MiB de almacenamiento a reservar para el volumen
```

Solución en la nube

Para desplegar la aplicación en la nube se utilizó DigitalOcean Inc. que es un proveedor de computación en la nube con sede en la ciudad de Nueva York y centros de datos en todo el mundo que ofrece una plataforma de infraestructura como servicio (IaaS) para desarrolladores de software. DigitalOcean es popular entre los desarrolladores de código abierto y compite con Amazon Web Services (AWS), entre otros.

Decidimos escoger esta opción, ya que entre los múltiples servicios, sus ofertas principales están destinadas a casos de uso de alojamiento de aplicaciones y sitios web, necesarias para nuestro proyecto.

Replicación de la arquitectura local en la nube:

La replicación de la arquitectura local en la nube implica trasladar una aplicación local a la nube manteniendo su funcionamiento, y con Docker obtenemos un gran ventaja, encapsulando la aplicación en contenedores, asegurando su portabilidad. Luego, con Kubernetes se orquestan esos contenedores en la nube, gestionando su escalabilidad y alta disponibilidad. Por último configuramos los recursos en la nube, como almacenamiento y bases de datos, para replicar la infraestructura original.

Los beneficios incluyen escalabilidad automática, alta disponibilidad, reducción de costos operativos y mayor agilidad. Así, se logra un entorno flexible y eficiente que replica la infraestructura local en la nube.

Configuración para el balanceador de carga para el backend replicado tres (3) veces:

En resumen, el balanceador de carga es un componente clave para distribuir el tráfico entre las réplicas del backend, garantizando alta disponibilidad y escalabilidad, no solo mejora el rendimiento y la eficiencia del sistema, sino porque también contribuye a una experiencia de usuario más robusta y satisfactoria.

1. Para la creación del balanceador de carga, seleccionamos desde el panel de control de DigitalOcean la opción de "Networking" y luego "Load Balancers".
2. Luego configuramos las reglas del balanceo, colocamos el protocolo, además del puerto de entrada y de salida
3. Después asignamos las réplicas para el backend al balanceador de carga mediante etiquetas.
4. También configuramos las comprobaciones de salud, la ruta, el intervalo, el tiempo de espera y número de intentos fallidos antes de considerar una réplica no disponible.

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

El uso de servicios en la nube, junto con la configuración de redes y seguridad, es fundamental para garantizar el rendimiento, la confiabilidad y la protección de las aplicaciones y datos en la nube. Estos servicios permiten optimizar la infraestructura, facilitar la conectividad y proteger los recursos de posibles amenazas.

Los servicios en la nube nos proporcionan una gama de recursos gestionados, como almacenamiento, bases de datos, y cómputo, que se pueden escalar según las necesidades de la aplicación. Utilizar estos servicios permite a las organizaciones evitar la gestión de infraestructura física, reduciendo costos y mejorando la eficiencia.

Por otro lado, la configuración adecuada de redes es esencial para asegurar una comunicación eficiente entre los servicios en la nube y con los usuarios. En la nube, las redes virtuales (VPC, Virtual Private Cloud) permiten crear una red privada dentro de la infraestructura pública, brindando control sobre la conectividad entre instancias, subredes y recursos.

La seguridad en la nube es un aspecto crítico para proteger datos y aplicaciones. Las plataformas en la nube proporcionan herramientas y servicios para mitigar riesgos y asegurar la integridad de los sistemas.

Descripción del flujo para subir la aplicación a la nube:

1. Preparación del proyecto: Verificamos que todo esté listo para desplegar la aplicación en DigitalOcean, Dokerfiles, docker-compose, kubernetes
2. Configuración del entorno en DigitalOcean: Creamos el cluster de kubernetes y subimos los archivos al cluster, verificando que el clúster tiene los nodos necesarios para manejar la carga y que las réplicas están distribuidas adecuadamente.
3. Subir la aplicación al servidor: Transferimos el código al servidor y subimos las imágenes al registro de contenedores en DigitalOcean
4. Configuración del balanceador de carga: Configuramos el balanceador de carga para distribuir el tráfico entre las réplicas del backend.
5. Pruebas y verificación: Accedemos al balanceador de carga a través de su IP o dominio configurado y realizamos pruebas funcionales para garantizar que las réplicas están recibiendo el tráfico correctamente.

Análisis

El *rendimiento* de la aplicación en un *ambiente local* muestra un tiempo de respuesta rápido debido a la proximidad del cliente y los recursos pero su capacidad, *fiabilidad* y *disponibilidad* es limitada debido al hardware disponible y la intervención humana para la solución de fallos o reasignación de recursos, por otra parte el *rendimiento en la nube* permite acceso a recursos más potentes y la *escalabilidad* automática de los contenedores según la demanda, gracias a Kubernetes manejando incrementos en la carga eficientemente y sin intervención manual, pero la latencia puede ser mayor debido a la distancia física entre el cliente y los servidores.

Uno de los *retos* fue tratar de integrar Docker de una manera correcta, ya que durante la construcción de los Dockerfile se tuvieron que hacer varias pruebas para encontrar una imagen que fuera ligera para que la aplicación tuviera buen rendimiento, también en la parte de Kubernetes fue difícil su implementación porque no se tenía el suficiente conocimiento, y solo *logramos salir adelante* haciendo pruebas paso a paso guiandonos con videos y ejemplos que mostraron aplicaciones similares.

Tenemos que *Docker* y *Kubernetes* son herramientas indispensables para desplegar y gestionar aplicaciones modernas; primero, porque *Docker* simplifica el empaquetado y transporte de aplicaciones, al garantizar que funcionen de manera consistente en cualquier entorno; segundo, porque *Kubernetes* amplió las capacidades de Docker al permitir la gestión avanzada de aplicaciones basadas en contenedores a gran escala con características como autoescalado y balanceo de carga.

El uso de Docker y Kubernetes para una aplicación web ofrece múltiples ventajas en términos de *escalabilidad*, *fiabilidad / disponibilidad* (mencionados anteriormente), *costo*, *optimización de recursos* y *seguridad*. En términos de *costo*, aunque hay costos iniciales de configuración y aprendizaje considerable, el uso de Docker y Kubernetes puede reducir los costos operativos a largo plazo mediante la optimización del uso de recursos y la capacidad de ejecutar aplicaciones en cualquier entorno de nube.

En cuanto a la *optimización de recursos*, Docker permite empaquetar aplicaciones con todas sus dependencias, asegurando que funcionen de manera consistente en diferentes entornos, y Kubernetes gestiona de manera eficiente los recursos del clúster, asignando y reubicando contenedores según sea necesario. Finalmente, en términos de *seguridad*, tanto Docker como Kubernetes soportan políticas de seguridad avanzadas, incluyendo la segmentación de contenedores y el cifrado de datos en tránsito y reposo.

Conclusiones

El proyecto del curso cumple el objetivo de aplicar los conocimientos adquiridos a lo largo del semestre como una base hacia lo que nosotros como desarrolladores decidamos que tenga nuestro propio lenguaje de programación, en sintaxis, semántica, características comunes o diferentes respecto a los lenguajes de programación existentes que cubra alguna necesidad ya sea de un cliente o mercado, o por el contrario crear un producto y/o servicio revolucionario en la industria.

Docker nos facilita la creación de entornos consistentes y portables, lo que elimina problemas de configuración entre equipos de desarrollo y producción.

Las plataformas en la nube proporcionan herramientas avanzadas para monitoreo, almacenamiento y balanceo de cargas que simplifican la administración de proyectos a gran escala. La combinación de ambas tecnologías resulta en una solución poderosa para el desarrollo ágil y el despliegue eficiente, mejorando tanto la experiencia del usuario como la competitividad de los equipos y empresas en un mercado tecnológico en constante evolución.