



Proyecto Final

Despliegue de aplicación con contenedores: Local y Nube

Infraestructuras paralelas y distribuidas

Elaborado por:

Rodas Arango, JUAN MANUEL - 2259571

García Castañeda, ALEX – 2259517

Gómez Agudelo, JUAN SEBASTIÁN – 2259474

Henao Aricapa, STIVEN – 2259603

Loaiza Serrano, JESÚS ESTENLLOS - 2313021

Docente:

Delgado Saavedra, CARLOS ANDRÉS

Sede Tuluá

Diciembre 2024

Índice

1. Introducción	3
1.1 Descripción proyecto.	3
1. 2 Objetivos del proyecto.	3
2. Solución Local Docker	4
2.1 Dockerfile backend	4
2.2 Dockerfile frontend	5
2.3 docker-compose.yml	6
2.4 Imágenes montadas	7
3. Solución Local Kubernetes	7
3.1 backend-config.yaml	7
3.2 backend-deployment.yaml	8
3.3 backend-secret.yaml	9
3.4 backend-service.yaml	10
3.5 frontend-deployment.yaml	11
3.6 frontend-service.yaml	12
3.7 Configuración kubectl apply back	13
3.8 Configuración kubectl apply front	13
4. Solución en la Nube	14
4.1 Creación Clúster de Kubernetes en Google Cloud con Autoscaling	14
5. Capturas funcionamiento aplicación	15
6. Análisis y Conclusiones	16
7. Conclusiones	17

1. Introducción

1.1 Descripción proyecto.

Fitfusion es una aplicación innovadora diseñada para ayudar a los usuarios a alcanzar sus metas de fitness de manera personalizada y eficiente. Adaptada a personas de cualquier nivel de experiencia, ofrece rutinas de entrenamiento predefinidas y personalizables que responden a las necesidades individuales. La aplicación permite a los usuarios crear un perfil personal con información física, preferencias y objetivos, generando automáticamente planes adaptados para perder peso, ganar músculo, mejorar la resistencia o mantener un estilo de vida saludable.

Además, Fitfusion incluye herramientas de seguimiento del progreso con estadísticas visuales y características de accesibilidad, como modo oscuro y ajustes de contraste, garantizando una experiencia inclusiva. Su diseño multiplataforma asegura compatibilidad con diversos dispositivos, y la aplicación está implementada tanto en entornos locales como en la nube, utilizando tecnologías modernas como Docker y Kubernetes, que aseguran escalabilidad y rendimiento óptimos.

1. 2 Objetivos del proyecto.

- **Ofrecer personalización en el entrenamiento:** Proveer rutinas adaptadas a las metas y capacidades de cada usuario.
- **Fomentar la accesibilidad:** Incluir funcionalidades que permitan a todas las personas, incluidas aquellas con discapacidades, utilizar la aplicación sin barreras.
- **Facilitar el seguimiento del progreso:** Implementar herramientas visuales que permitan a los usuarios monitorear su desempeño y ajustar sus objetivos.
- **Promover un estilo de vida saludable:** Ser una solución integral que motive a los usuarios a mejorar su bienestar físico de forma constante y sostenible.
- **Garantizar compatibilidad multiplataforma:** Asegurar que la aplicación funcione correctamente en diferentes dispositivos y sistemas operativos para llegar a un público amplio.
- **Optimizar el despliegue:** Implementar la aplicación en entornos locales y en la nube mediante Docker y Kubernetes, garantizando escalabilidad, rendimiento y facilidad de mantenimiento.

2. Solución Local Docker

2.1 Dockerfile backend

```
1      # Usar la imagen base de Node.js
2      FROM node:18-alpine
3
4      # Establecer el directorio de trabajo en /app
5      WORKDIR /app
6
7      # Copiar los archivos de dependencias (package.json y package-lock.json)
8      COPY package*.json ./
9
10     # Instalar todas las dependencias (incluyendo dependencias de desarrollo)
11     RUN npm install
12
13     # Copiar el resto de los archivos del proyecto al contenedor
14     COPY . /app
15
16     # Ejecutar la compilación
17     RUN npm run build
18
19     # Exponer el puerto 3000
20     EXPOSE 3000
21
22     # Establecer el comando para iniciar la aplicación
23     CMD ["npm", "run", "start"]
```

Este Dockerfile configura un contenedor para el backend de Fitfusion utilizando Node.js.

Se basa en una imagen ligera de Node.js (**node:18-alpine**) que reduce el tamaño del contenedor. Establece **/app** como directorio de trabajo y copia los archivos **package.json** y **package-lock.json** al contenedor para instalar las dependencias con **npm install**. Luego, copia todos los archivos del proyecto al contenedor y ejecuta el comando **npm run build**, que compila el backend (por ejemplo, para generar código optimizado). Expone el puerto **3000** para que la aplicación esté accesible desde ese puerto y define el comando **npm run start** como el punto de entrada del contenedor para arrancar la aplicación.

2.2 Dockerfile frontend

```
1  FROM node:18 AS build
2  WORKDIR /app
3  COPY package.json package-lock.json ./
4  RUN npm install
5  COPY . .
6  RUN npm run build
7  FROM node:18 AS production
8  WORKDIR /app
9  COPY --from=build /app/package.json /app/package-lock.json ./
10 RUN npm install --only=production
11 COPY --from=build /app/dist ./dist
12 EXPOSE 3000
13 CMD ["npm", "run", "preview", "--", "--host", "0.0.0.0"]
```

Este Dockerfile define dos fases: una para construir (*build*) y otra para producción (*production*).

Fase de construcción: Usa la imagen de Node.js para instalar las dependencias del frontend (npm install) y ejecutar npm run build, que genera los archivos optimizados para producción en la carpeta dist.

Fase de producción: Usa una nueva imagen de Node.js y copia únicamente los archivos necesarios de la fase anterior (package.json, package-lock.json y dist) para mantener el contenedor más ligero. Se instalan solo las dependencias de producción con npm install --only=production. Finalmente, el contenedor expone el puerto 3000 y utiliza npm run preview para servir el frontend optimizado, asegurando que esté disponible en todas las interfaces (--host 0.0.0.0).

2.3 docker-compose.yml

```
# docker-compose.yml
1 services:
2   backend:
3     build: ./Backend-FitFusion/
4     ports:
5       - "3000:3000"
6     environment:
7       - MONGO_URI=mongodb+srv://dev:DEV@fitfusioncluster0.3wwaa.mongodb.net/?retryWrites=true&w=majority&appName=FitFusionCluster0
8       - JWT_SECRET=tu_secreto_jwt
9       - GOOGLE_CLIENT_ID=63435502995-3d5utqrb6nkqje7hkr7apn8rr7l6i2qv.apps.googleusercontent.com
10      - GOOGLE_CLIENT_SECRET=GOCSNX-J5Dni0YDmfx_3PumVMOk5r6wha_C
11      - GOOGLE_REDIRECT_URL=http://localhost:3000/api/auth/google/callback
12      - JWT_EXPIRES_IN=1h
13     networks:
14       - app_network
15
16   frontend:
17     build: ./Frontend-fitfusion/
18
19     ports:
20       - "5173:5173"
21     environment:
22       - VITE_API_URL=http://backend:3000
23     networks:
24       - app_network
25
26   networks:
27     app_network:
28       driver: bridge
```

Este archivo configura el entorno para ejecutar los servicios de frontend y backend de Fitfusion utilizando Docker Compose.

Servicio backend

- Construye una imagen para el backend desde el directorio ./Backend-FitFusion/.
- Mapea el puerto interno 3000 al puerto 3000 del host, permitiendo acceso externo al backend.

- Define varias variables de entorno necesarias para la configuración:

- **MONGO_URI**: URL de la base de datos MongoDB.

- **JWT_SECRET**: Clave secreta para firmar tokens JWT. Configuraciones de autenticación con Google (ID, secreto y URL de redirección).

- Tiempo de expiración de los tokens JWT.

- Asigna el backend a la red app_network para permitir comunicación con el frontend.

Servicio frontend:

- Construye una imagen para el frontend desde el directorio ./Frontend-fitfusion/.
- Mapea el puerto interno 5173 al puerto 5173 del host, haciendo accesible el frontend.
- Define una variable de entorno VITE_API_URL para que el frontend pueda comunicarse con el backend utilizando el nombre del servicio backend como host.
- También se conecta a la red app_network.

Red app_network:

- Define una red tipo bridge, que permite que los servicios frontend y backend se comuniquen entre sí directamente mediante sus nombres.

2.4 Imágenes montadas

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
proyectofitfusion-frontend	latest	7f082e6d8553	29 seconds ago	1.94GB
proyectofitfusion-backend	latest	994c09f906ec	About a minute ago	375MB

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
8ab2bf4e3ca	proyectofitfusion-backend	"docker-entrypoint.s..."	43 seconds ago	Up 40 seconds	0.0.0.0:3000->3000/tcp	proyectofitfusion-backend-1
7ee61e0d9d51	proyectofitfusion-frontend	"docker-entrypoint.s..."	43 seconds ago	Up 40 seconds	3000/tcp, 0.0.0.0:5173->5173/tcp	proyectofitfusion-frontend-1

3. Solución Local Kubernetes

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: backend-config
5  data:
6    GOOGLE_REDIRECT_URI: "http://localhost:3000/api/auth/google/callback"
7    JWT_EXPIRES_IN: "1h"
```

3.1 backend-config.yaml

Este archivo define un **ConfigMap**, un recurso de Kubernetes utilizado para almacenar configuraciones en forma de pares clave-valor.

- Contiene configuraciones no sensibles que el backend necesita para funcionar, como el tiempo de expiración del token JWT (**JWT_EXPIRES_IN**) y el URI de redirección de Google (**GOOGLE_REDIRECT_URI**).
- Este ConfigMap puede ser referenciado en los contenedores del backend para injectar estas variables de configuración como variables de entorno.

3.2 backend-deployment.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: backend-deployment
5    labels:
6      app: backend
7  spec:
8    replicas: 3
9    selector:
10      matchLabels:
11        app: backend
12    template:
13      metadata:
14        labels:
15          app: backend
16    spec:
17      containers:
18        - name: backend
19          image: manuelarango1229/backfitfusion:latest
20          ports:
21            - containerPort: 3000
22          envFrom:
23            - configMapRef:
24              name: backend-config
25            - secretRef:
26              name: backend-secrets
```

Este archivo define un **Deployment**, que administra y asegura la ejecución de múltiples réplicas del backend.

- **replicas: 3**: Kubernetes desplegará 3 pods del backend para mejorar la disponibilidad y capacidad de respuesta.
- **Contenedor principal**:
 - Utiliza la imagen manuelarango1229/backfitfusion:latest.
 - Expone el puerto 3000 dentro de cada pod.
 - Inyecta configuraciones desde el **ConfigMap** (backend-config) y secretos desde el **Secret** (backend-secrets) como variables de entorno.
- **Labels y Selectors**: Asocian los pods creados con los servicios mediante la etiqueta app: backend.

3.3 backend-secret.yaml

```
1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: backend-secrets
5   type: Opaque
6 data:
7   MONGO_URI: bw9uZ9kYitzcnY6Ly9kZXy6REWQGZpdGZ1c21vbmmNsdxN0ZXIwLjN3d2FhLm1vbmdvZGIubmV0Lz9yZXByeVdyaxRlcz10cnVlJnc9bwFqb3JpdHkmYXBwTmFtZT1GaXRGdXNpb25DbHVzdGVyMA==
8   JWT_SECRET: dHfc2VjcmV0b19qd3Q=
9   GOOGLE_CLIENT_ID: NjM0MzU1MDI5OTUtM2Q1dXRxcmI2bmsqamU3aGtyN2Fwbg==
10  GOOGLE_CLIENT_SECRET: R09DU1BYLUUpTRE5phF1EbWz4xzNQdw1MTU9rNVI2d2hhX0M=
```

Este archivo define un **Secret**, un recurso de Kubernetes utilizado para almacenar información confidencial en forma codificada en base64.

- Contiene valores sensibles necesarios para el backend, como:
 - La URI de conexión a MongoDB (**MONGO_URI**).
 - La clave secreta para los tokens JWT (**JWT_SECRET**).
 - Credenciales para la autenticación con Google (**GOOGLE_CLIENT_ID** y **GOOGLE_CLIENT_SECRET**).
- Los datos en el archivo están codificados en base64 para su transmisión segura dentro del clúster de Kubernetes.
- Estos secretos son referenciados en el Deployment para ser usados como variables de entorno en los contenedores.

3.4 backend-service.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: backend-service
5  spec:
6    type: LoadBalancer
7    selector:
8      app: backend
9    ports:
10      - protocol: TCP
11        port: 3000
12        targetPort: 3000
13        nodePort: 30001
```

Este archivo define un **Service**, un recurso de Kubernetes que expone los pods del backend para que puedan ser accedidos internamente o externamente.

- **type: LoadBalancer**: Permite exponer el servicio fuera del clúster, distribuyendo las solicitudes entrantes entre los pods del backend mediante un balanceador de carga.
- **Selector**: Conecta el servicio con los pods etiquetados como `app: backend` creados por el Deployment.
- **Puertos**:
 - **port**: Puerto expuesto por el servicio (3000).
 - **targetPort**: Puerto interno donde escucha el contenedor del backend (3000).
 - **nodePort**: Puerto asignado en los nodos del clúster para acceder al servicio (30001).

3.5 frontend-deployment.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: frontend-deployment
5    labels:
6      app: frontend
7  spec:
8    replicas: 1
9    selector:
10      matchLabels:
11        app: frontend
12      template:
13        metadata:
14          labels:
15            app: frontend
16        spec:
17          containers:
18            - name: frontend
19              image: manuelarango1229/frontfitfusion:latest
20              ports:
21                - containerPort: 5173
22              env:
23                - name: VITE_API_URL
24                  value: "http://192.168.49.2:30001"
```

Este archivo define un **Deployment** para el frontend de Fitfusion.

- **replicas: 1:** Se despliega solo un pod del frontend, ya que típicamente los servicios frontend requieren menos escalabilidad inicial comparado con el backend.
- **Contenedor principal:**
 - Usa la imagen manuelarango1229/frontfitfusion:latest, que contiene el frontend de Fitfusion.
 - Expone el puerto interno 5173, donde el frontend está configurado para servir la aplicación.
 - Inyecta una variable de entorno VITE_API_URL con el valor http://192.168.49.2:30001. Esto indica que el frontend se comunicará con el backend mediante el puerto expuesto por el backend-service.
- **Labels y Selectors:**
 - La etiqueta app: frontend identifica los pods creados por este Deployment, permitiendo que otros recursos (como servicios) los encuentren.

3.6 frontend-service.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: frontend-service
5  spec:
6    selector:
7      app: frontend
8    ports:
9      - protocol: TCP
10     port: 80
11     targetPort: 4173
12     nodePort: 30968
13   type: LoadBalancer
```

Este archivo define un **Service** que expone el frontend para que pueda ser accedido externamente.

- **type: LoadBalancer**: Expone el servicio fuera del clúster mediante un balanceador de carga. Esto permite que los usuarios accedan al frontend desde internet o redes externas.
- **Selector**: Conecta el servicio con los pods etiquetados como app: frontend creados por el Deployment.
- **Puertos**:
 - **port**: El puerto 80 es el puerto estándar HTTP que se expone para el acceso externo.
 - **targetPort**: Redirige el tráfico al puerto interno 4173 del pod del frontend (configuración del contenedor para servir la aplicación).
 - **nodePort**: Permite acceder al servicio desde el puerto 30968 de los nodos del clúster.

Este **Service** asegura que el frontend sea accesible desde fuera del clúster, utilizando un balanceador de carga para redirigir el tráfico al pod correspondiente.

3.7 Configuración kubectl apply back

```
Backend-FitFusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co
⌚ 6% > ls
{ }backend-config.yaml { }backend-deployment.yaml { }backend-secret.yaml { }backend-service.yaml

Backend-FitFusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co
⌚ 6% > nvim backend-service.yaml

Backend-FitFusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co took 22s
⌚ 7% > kubectl apply -f backend-config.yaml
configmap/backend-config created

Backend-FitFusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co took 3s
⌚ 8% > kubectl apply -f backend-secrets.yaml
backend-secret.yaml backend-service.yaml

Backend-FitFusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co took 3s
⌚ 8% > kubectl apply -f backend-service.yaml
secret/backend-secrets created

Backend-FitFusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co
⌚ 8% > kubectl apply -f backend-deployment.yaml
deployment.apps/backend-deployment created

Backend-FitFusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co
⌚ 8% > kubectl apply -f backend-service.yaml
service/backend-service created

Backend-FitFusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co
⌚ 9% > |
```

3.8 Configuración kubectl apply front

```
Frontend-fitfusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co
⌚ 9% > ls
{ }frontend-deployment.yaml { }frontend-service.yaml

Frontend-fitfusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co
⌚ 9% > kubectl apply -f frontend-deployment.yaml
deployment.apps/frontend-deployment created

Frontend-fitfusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co
⌚ 10% > kubectl apply -f frontend-service.yaml
service/frontend-service created

Frontend-fitfusion/k8s on ℹ main [!?] on ~ juan.arango.rodas@correounalvalle.edu.co
⌚ 10% > |
```

3.9 Visualización pods corriendo, IP externa y get service

```
Backend-FitFusion/k8s on ℹ main [!?]
> kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
backend-deployment-888b8bd5f-465dv   1/1     Running   0          15s
backend-deployment-888b8bd5f-gmw8s   1/1     Running   0          15s
backend-deployment-888b8bd5f-qftwf   1/1     Running   0          15s
frontend-deployment-646699bf65-8zsr4  1/1     Running   0          4m47s

Backend-FitFusion/k8s on ℹ main [!?]
> kubectl get service
NAME         TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
backend-service   NodePort   10.96.122.178 <none>        3000:30001/TCP   14s
frontend-service  NodePort   10.102.18.132  <none>        80:30968/TCP   4m47s
kubernetes       ClusterIP  10.96.0.1    <none>        443/TCP      5m27s
```

4. Solución en la Nube

4.1 Creación Clúster de Kubernetes en Google Cloud con Autoscaling

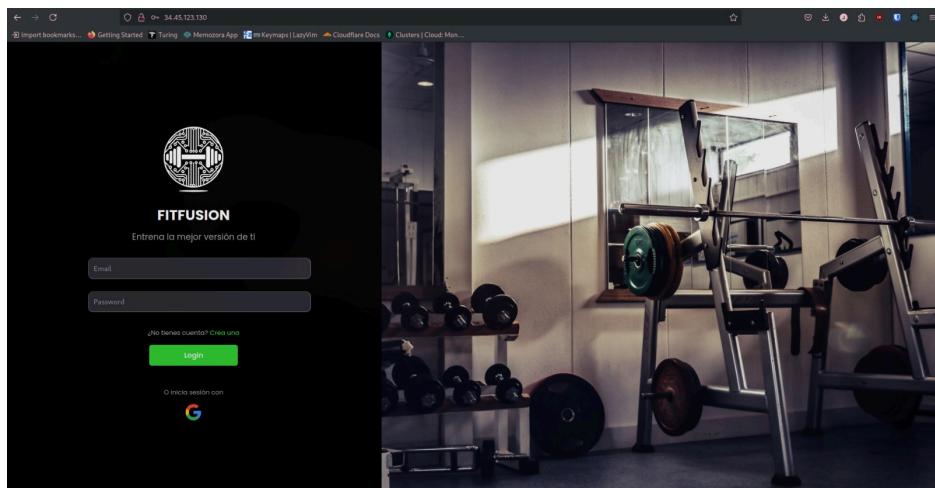
```
~ via ~ on ~ juan.arango.rodas@correounivalle.edu.co
> gcloud container clusters create fitfusion-cluster --region us-central1 --num-nodes 1 --machine-type e2-micro --disk-size 30 --enable-autoscaling --min-nodes 1 --max-nodes 3
```

- a. **gcloud container clusters create fitfusion-cluster:**
Crea un clúster de Kubernetes llamado fitfusion-cluster.
- b. **--region us-central1:**
Especifica que el clúster estará alojado en la región us-central1 (centro de Estados Unidos).
- c. **--num-nodes 1:**
Inicializa el clúster con un nodo activo.
- d. **--machine-type e2-micro:**
Define el tipo de máquina virtual (VM) para los nodos del clúster. El tipo e2-micro es económico y tiene recursos limitados (ideal para pruebas o entornos de desarrollo).
- e. **--disk-size 30:**
Configura un disco de 30 GB para el nodo. Esto es útil para almacenar datos temporales o persistentes en las máquinas virtuales.
- f. **--enable-autoscaling:**
Habilita el escalado automático de nodos según la carga del clúster.
- g. **--min-nodes 1 --max-nodes 3:**
Configura los límites de autoscaling:
 - Mínimo: 1 nodo.
 - Máximo: 3 nodos.

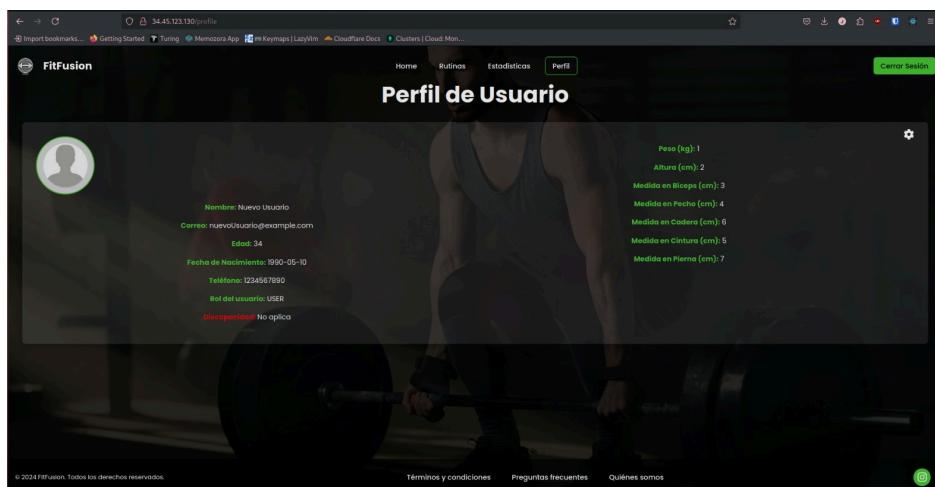
DESCRIPCIÓN GENERAL		OBSERVABILIDAD		OPTIMIZACIÓN DE COSTOS			
Filtro		Ingresar el nombre o el valor de la propiedad					
Estado	Nombre	Ubicación	Nivel	Cantidad de nodos	CPU virtuales totales	Memoria total	Notificaciones
<input type="checkbox"/>	<input checked="" type="checkbox"/> fitfusion-cluster	us-central1	Estándar	4	8	4 GB	⚠️ Poda no programables ⚠️ No se pueden escalar verticalmente los nodos

Visibilidad de clúster en funcionamiento

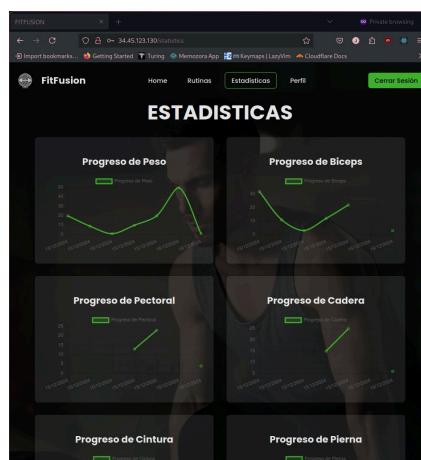
5. Capturas funcionamiento aplicación



Aplicación funcionando en la nube con IP externa



El redireccionamiento se ha ejecutado correctamente y los datos del usuario se han obtenido con éxito, lo que confirma que el backend está funcionando de manera adecuada.



Más demostración de aplicación en funcionamiento

6. Análisis y Conclusiones

Rendimiento de la Aplicación: Ambiente Local vs. Nube

El análisis del rendimiento de la aplicación en un entorno local y en la nube mostró diferencias significativas en latencia, escalabilidad y disponibilidad. En el entorno local, la latencia era más baja debido a la proximidad de los recursos. Sin embargo, la capacidad de escalabilidad era limitada por las restricciones del hardware físico. En la nube, la latencia era ligeramente mayor debido al acceso remoto a los servicios, pero la escalabilidad y disponibilidad del sistema fueron superiores gracias a las capacidades de autoscaling y balanceo de carga proporcionadas por Kubernetes.

Latencia, Escalabilidad y Disponibilidad del Sistema

- **Latencia:** En la nube, la latencia depende de la ubicación geográfica del usuario y del clúster. Aunque se observó un incremento marginal en comparación con el entorno local, el uso de servicios como Load Balancer ayudó a mantenerla en niveles aceptables.
- **Escalabilidad:** Kubernetes permitió ajustar los recursos dinámicamente, aumentando el número de réplicas del backend según la carga. Esto es crítico para manejar picos de tráfico.
- **Disponibilidad:** Gracias a la implementación de múltiples réplicas y estrategias de recuperación, el sistema mantuvo su funcionalidad incluso en caso de fallas de nodos.

Retos Abordados

- Configuración de Kubernetes para el despliegue automático y seguro de aplicaciones.
- Manejo de secretos y variables sensibles usando ConfigMaps y Secrets.
- Optimización del consumo de recursos mediante el uso de instancias de bajo costo con autoscaling en la nube.
- Reducción de latencia mediante la implementación de un balanceador de carga y estrategias de caché en el frontend.

Reflexiones sobre Docker y Kubernetes

El uso de Docker simplificó la creación de imágenes consistentes para los entornos de desarrollo, pruebas y producción. Kubernetes proporcionó un marco robusto para la orquestación de contenedores, asegurando escalabilidad, alta disponibilidad y gestión eficiente de los recursos.

7. Conclusiones

El desarrollo y despliegue del proyecto en la nube utilizando tecnologías como Docker y Kubernetes proporcionaron aprendizajes valiosos en términos de gestión de infraestructura moderna.

- 1. Escalabilidad:** La capacidad de ajustar los recursos dinámicamente es fundamental para manejar cargas variables y asegurar un rendimiento óptimo.
- 2. Fiabilidad:** La implementación de múltiples réplicas y estrategias de recuperación automática garantizó la continuidad del servicio incluso en condiciones adversas.
- 3. Costo:** Aunque los costos iniciales de configuración en la nube pueden parecer elevados, el ahorro a largo plazo y la flexibilidad compensan ampliamente la inversión.
- 4. Optimización de Recursos:** Se logró un uso eficiente de recursos al implementar autoscaling, configurando correctamente los límites de CPU y memoria, y evitando recursos infrautilizados.
- 5. Seguridad:** El manejo adecuado de secretos y configuraciones sensibles demostró ser clave para proteger la aplicación y los datos.

La experiencia obtenida tras realizar el despliegue de la aplicación con contenedores en local y nube demostró que tecnologías como Docker y Kubernetes son herramientas poderosas para construir sistemas modernos y escalables. Sin embargo, también resaltó la importancia de planificar adecuadamente, monitorear el rendimiento y optimizar los recursos para obtener un balance entre costo y rendimiento.

El diagrama de arquitectura propuesto ilustra cómo se gestionaron y comunicaron los componentes Frontend, Backend y Base de Datos, destacando la separación de responsabilidades y el uso de servicios en la nube para maximizar la eficiencia.