PRACTICA INTEGRADORA: HERRAMIENTAS INFORMÁTICAS AVANZADAS

OBJETIVO: Este trabajo práctico integrador está diseñado para evaluar el manejo de herramientas avanzadas en la implementación, despliegue y mantenimiento de aplicaciones web.

Alumnos Integrantes

APU005951 Alarcón Arias, Marisel Evelin

APU006031 Martinez Gonzalez, Maximiliano

APU006313 Pelazzo, Maximiliano Alejandro

APU006053 Sivila, Fernanda Belén

1. Selección de la aplicación web

Tarea: Seleccionar una aplicación web previamente desarrollada (por ejemplo, un proyecto académico). Identificar el stack tecnológico utilizado:

- **Backend: Node.js, Spring, Django, etc.**
- **Frontend: Angular, React, etc.**
- **Base de datos: PostgreSQL, MySQL, MongoDB, etc. o**
- **Si la aplicación no está en un repositorio remoto, utiliza Git para inicializar y subirla (GitHub, GitLab, etc.).**

Resolve Gym

Aplicación web realizada como proyecto final de la materia Programación y Servicios Informáticos

Resolve es un sistema integral de gestión para gimnasios, diseñado para facilitar la administración de socios, empleados, accesorios y actividades. Este sistema permite a los usuarios realizar un seguimiento eficiente de las operaciones diarias y mejorar la experiencia tanto para el personal administrativo como para los socios del gimnasio.

Stack Tecnológico utilizado

- **Frontend:** Angular, HTML, CSS, Bootstrap, TypeScript
- **Backend:** Node.js, Express.js
- **Base de Datos:** MongoDB

Sistema-de-Gestión-de-Gimnasios**.**

2. Actividad 1: Contenedores para servicios y base de datos

Objetivo: *Dockerizar la aplicación. Para esto…*

---

1. **Base de datos:**
    - **Crear un contenedor para la base de datos (p.ej., PostgreSQL o MySQL).**
    - **Configurar un cliente dockerizado (p.ej., pgAdmin para PostgreSQL o phpMyAdmin para MySQL).**
    - **Migrar los datos existentes al contenedor.**
    - **Backups automatizados: Configura un contenedor adicional para ejecutar backups automáticos de la base de datos. Usa herramientas como pg_dump (PostgreSQL) o mysqldump (MySQL).**
2. **Recomendación: Utiliza un archivo docker-compose.yml para gestionar la base de datos y el cliente.**

Para asegurar que los contenedores puedan comunicarse entre ellos sin exponer de forma innecesaria los servicios al exterior, se creó una red personalizada llamada: **mongo_cluster**.

También configuramos un rango de direcciones IP personalizado (`192.168.0.0/16`) para evitar conflictos de red.

Contenedor `mongo_db`

El primer paso fue crear un contenedor para alojar la base de datos. Utilizamos la imagen oficial de **MongoDB**, ya que es ligera y funcional. Este contenedor se encarga de alojar y gestionar los datos que utiliza la aplicación.

Ya que la base de datos de nuestra aplicación Resolve Gym se encontraba en la nube (Mongo Atlas) fue necesario importar los datos:

```
mongodump
--uri="mongodb+srv://resolvegym10:<password>@gym-db.g9wzrka.mongodb.net/"
--out="Descargas"
```

`mongodump` es una **herramienta que viene incluida en la instalación de MongoDB**. Se ejecuta desde la línea de comandos en la terminal. Se utiliza para **crear copias de seguridad de bases de datos de MongoDB**. Permite exportar el contenido de una o varias bases de datos en formato binario BSON (Binary Json).

La ejecución del comando me generó la carpeta gym_db, que **contiene los BSON de todas las colecciones que implementamos.**

Dentro del contenedor configuramos un usuario administrador (admin) y una contraseña (admin1234), utilizando **variables de entorno**. Esto asegura que la base de datos **esté protegida y que solo usuarios autorizados puedan acceder.**

Para mayor organización se creó la carpeta `mongo_data` que contiene la carpeta `gym_db` y `mongo_script`. Esta se utilizaron al implementar **volúmenes**:

- **Carga de datos**: `./mongo_data/gym_db:/data/dump`. Esto posibilita la carga automática de datos existentes al iniciar el contenedor.
- **Script de inicialización**: `./mongo_data/mongo_script:/docker-entrypoint-initdb.d`. Al arrancar el contenedor, MongoDB ejecuta un script llamado `init-mongo.sh` que restaura automáticamente los datos preexistentes en la base de datos (que se encuentran en gym_db).

```
#!/bin/bash
mongorestore --drop --db=gym_db --dir=/data/dump
```

Este script utiliza `mongorestore`, una herramienta que permite cargar datos en una instancia de MongoDB desde un volcado de base de datos binario o desde la entrada estándar. Puede restaurar archivos BSON generados por mongodump.

El puerto `27017` fue mapeado para que otros servicios o herramientas (como el cliente gráfico) puedan comunicarse con la base de datos.

```
mongo_db:
    image: mongo:latest
    container_name: mongo_db
    ports:
      - "27017:27017"
    volumes:
      - ./mongo_data/gym_db:/data/dump
      - ./mongo_data/mongo_script:/docker-entrypoint-initdb.d
      - ./mongo_backups:/backups
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: admin1234
    restart: always
    networks:
      - mongo-cluster
```

Contenedor `mongo-cliente`

El siguiente paso fue configurar un cliente gráfico para interactuar con la base de datos. **Mongo Express** es un cliente liviano que permite administrar MongoDB desde un navegador web.

Este contenedor es útil para realizar tareas administrativas como:

- Visualizar datos.
- Modificar registros.
- Ejecutar consultas rápidas.

El mongo-express se configuró para conectarse al contenedor de la base de datos utilizando las credenciales de administrador (`admin` y `admin1234`) y la URL de conexión (**`mongodb://admin:admin1234@mongo_db:27017/gym_db?authSource=admin`**). Para proteger el acceso, agregamos autenticación básica con un usuario (**`user`**) y contraseña (**`admin1234`**).

El puerto **`8081`** fue mapeado para que la interfaz gráfica esté disponible desde el navegador. De esta forma, se puede acceder fácilmente desde `http://localhost:8081`.

```
mongo-client:
    image: mongo-express:latest
    container_name: mongo_client
    ports:
      - "8081:8081"
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: admin
      ME_CONFIG_MONGODB_ADMINPASSWORD: admin1234
      ME_CONFIG_MONGODB_ENABLE_ADMIN: true
      ME_CONFIG_BASICAUTH_USERNAME: user
      ME_CONFIG_BASICAUTH_PASSWORD: admin1234
      ME_CONFIG_MONGODB_URL:
mongodb://admin:admin1234@mongo_db:27017/gym_db?authSource=admin
    restart: always
    networks:
      - mongo-cluster
```

Contenedor `mongo-backup`

Este contenedor fue diseñado para **automatizar la creación de copias de seguridad de la base de datos**. Es especialmente útil para garantizar que **no se pierdan datos importantes en caso de fallos.**

Para este contenedor, construimos una imagen personalizada basada en **Alpine Linux**, que es ligera y eficiente. En esta imagen, instalamos las herramientas necesarias (`mongodb-tools`) para realizar los backups, entre ellas están tanto el mongodump como el mongostore, y se implementa el script backup.sh.

```
FROM alpine:latest

RUN apk update && apk add --no-cache mongodb-tools bash

RUN mkdir -p /backups

COPY backup.sh /usr/local/bin/backup.sh
```

```
RUN chmod +x /usr/local/bin/backup.sh

CMD ["/usr/local/bin/backup.sh"]
```

El script `backup.sh`, que se copia dentro del contenedor, realiza las siguientes acciones:

1. Cada 30 minutos, ejecuta el comando `mongodump` para crear un volcado completo de la base de datos.
2. El backup se guarda en un archivo comprimido (`backup_<fecha>.gz`) dentro de la carpeta `./mongo_backups/backups`. Esto facilita la gestión de copias de seguridad y ahorra espacio en disco.

```
#!/bin/sh
# Ruta del directorio de backups
BACKUP_DIR="/backups"
mkdir -p $BACKUP_DIR
sleep 10
while true; do
  TIMESTAMP=$(date '+%Y-%m-%d_%H-%M-%S')
  mongodump --host=mongo_db --port=27017 --db=gym_db --username=admin
--password=admin1234 --authenticationDatabase=admin
--authenticationMechanism=SCRAM-SHA-1 --archive=$BACKUP_DIR/backup_$TIMESTAMP.gz
  echo "Backup creado en $BACKUP_DIR/backup_$TIMESTAMP.gz"
  sleep 1800
done
```

También se implementó un volumen:

Almacenar backups: ./mongo_backups/backups:/backups. Para mayor organización se creó una carpeta llamada mongo_backups, está contiene la carpeta backups, que almacenará las mismas, el script backup.sh y Dockerfile.

```
mongo-backup:
    build:
      context: ./mongo_backups
      dockerfile: Dockerfile
    container_name: mongo_backup
    volumes:
      - ./mongo_backups/backups:/backups
    depends_on:
      - mongo_db
    networks:
      - mongo-cluster
```

3. Actividad 2: Contenedores para servicios web

Objetivo: *Dockerizar la aplicación principal (backend, frontend) y exponerla para que sea accesible desde el navegador.*

- Configura contenedores para cada componente:
    - Frontend: Angular/React.
    - Backend: Node.js/Spring.
- Despliegue: Configura los puertos y dependencias en el docker-compose.yml.
- Pruebas: Asegúrate de que la aplicación funcione correctamente con datos mínimos cargados.
- Integre herramientas como Prometheus y Grafana para monitorear métricas en tiempo real (CPU, memoria, etc.).

FRONTEND

Para el Frontend de nuestra aplicación web, estamos utilizando la tecnología de Angular (Proyecto de Programación y Servicios Web).

1. Descargar del repositorio la sección de Frontend. En nuestro caso, dividimos tanto el backend como el frontend en ramas distintas por lo que hacer git clone de la rama "Frontend", fue suficiente.

2. Crear un archivo Dockerfile:

```
FROM node:18-alpine AS build
RUN mkdir -p /app
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
RUN npm run build --prod

FROM nginx:1.17.1-alpine
COPY nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build /app/dist/resolve-gym-website /usr/share/nginx/html/assets
```

Construcción: En la primera etapa, el Dockerfile compila la aplicación web en un entorno optimizado para producción.

Despliegue: En la segunda etapa, la imagen de Nginx se usa para servir los archivos estáticos generados.

3- Crear archivo de configuración de Nginx: Tenemos que crear un archivo de configuración para poder mostrar nuestra aplicación apenas se inicie el contenedor con Nginx. El archivo se debe llamar nginx.conf y luego lo utilizaremos como en un volumen en el docker-compose:

```
server {
    listen       4200;
    server_name  localhost;

    #charset koi8-r;
    #access_log  /var/log/nginx/host.access.log  main;

    location / {
        root   /usr/share/nginx/html/assets/browser/;
        index  index.html index.htm;
    }

    #error_page  404              /404.html;

    # redirect server error pages to the static page /50x.html
    #
    error_page   500 502 503 504  /50x.html;
```

```
    location = /50x.html {
        root   /usr/share/nginx/html;
    }

    # proxy the PHP scripts to Apache listening on 127.0.0.1:80
    #
    #location ~ \.php$ {
    #    proxy_pass   http://127.0.0.1;
    #}

    # pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000
    #
    #location ~ \.php$ {
    #    root           html;
    #    fastcgi_pass   127.0.0.1:9000;
    #    fastcgi_index  index.php;
    #    fastcgi_param  SCRIPT_FILENAME  /scripts$fastcgi_script_name;
    #    include        fastcgi_params;
    #}

    # deny access to .htaccess files, if Apache's document root
    # concurs with nginx's one
    #
    #location ~ /\.ht {
    #    deny  all;
    #}
}
```

4- Configurar el docker-compose:

```
frontend:
    image: maxmartinez29/docker-resolve-frontend:latest
    container_name: frontend
    ports:
      - "4200:4200"
    depends_on:
      - mongo_db
    restart: always
    networks:
      - mongo-cluster
```

Este archivo define la configuración del servicio `frontend`, construyendo la imagen desde un Dockerfile y configurando el servidor Nginx para servir la aplicación web. También asegura que:

- La aplicación funciona en el puerto 4200.
- El contenedor utiliza una configuración personalizada de Nginx.
- Se reinicia automáticamente si falla.
- Se conecta a una red llamada `mongo-cluster` y espera que la base de datos MongoDB esté disponible antes de iniciar.

Una vez realizado todos estos pasos, tendremos un Frontend funcionando en el puerto http://localhost:4200.

BACKEND

El proceso para configurar el backend es similar al del frontend. A continuación se detallan los pasos para configurar el entorno del backend utilizando Docker y conectar la base de datos MongoDB.

1. **Descargar el Proyecto:**

   Descargamos el proyecto de servicios web desde el repositorio correspondiente y lo colocamos en una carpeta llamada Backend.

2. **Dockerfile:**

   Dentro de la carpeta Backend, creamos un archivo Dockerfile con el siguiente contenido:

```
FROM node:20
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD [ "npm","start" ]
```

La imagen base para nuestro contenedor será la imagen oficial de Node.js, específicamente la versión 20. Luego se establece el directorio de trabajo dentro del contenedor "/app" donde se copian los archivos **.json* que contienen las dependencias necesarias para la aplicación.

Se copian todos los archivos de la carpeta local (excepto los que están en .dockerignore) al contenedor, dentro del directorio de trabajo (/app).

Y en la última línea se indica que cuando el contenedor inicie, se ejecutará npm start, que inicia la aplicación Node.js definida en el *package.json*.

```
"scripts": {
    "start": "node src/app.js",
```

3. **Archivo .dockerignore:**

Para evitar la inclusión de archivos innecesarios, configuramos un archivo *.dockerignore* que contiene:

```
node_modules
# Ignorar archivos de logs
npm-debug.log
*.log
# Ignorar archivos del sistema
.DS_Store
Thumbs.db
```

4. **Configuración de Docker Compose**

En el archivo docker-compose.yml, configuramos los siguientes parámetros:

```
backend:
    container_name: backend
    build:
      context: ./Backend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      -
MONGO_URI=mongodb://admin:admin1234@mongo_db:27017/gym_db?authSource=admin
    depends_on:
      - mongo_db
    restart: always
    networks:
      - mongo-cluster
```

Se define el contexto de construcción del contenedor, que es la carpeta Backend donde se encuentra el Dockerfile. Exponemos el puerto 3000 para tener acceso al contenedor en la máquina local.

Definimos una variable de entorno, *MONGO_URI*, que contiene la URL de conexión a la base de datos MongoDB. El contenedor del backend utilizará esta variable para conectarse a la base de datos.

5. **Conexión a la Base de Datos**

En el archivo *database*.js, configuramos la conexión a la base de datos MongoDB utilizando la variable de entorno definida previamente:

```
const mongoose = require("mongoose");
const URI = process.env.MONGO_URI
```

6. **Lógica de Reintentos para Conexión**

Dado que el contenedor backend depende del contenedor mongo_db (que puede tardar en inicializarse debido a su tamaño), y si bien declaramos esto con ‘depends_on’ en el archivo compose, no se garantiza que Docker inicie primero el contenedor de MongoDB antes de iniciar el contenedor de backend.

Por ello, en el archivo *database*.js, se utiliza el siguiente código para gestionar la conexión a la base de datos MongoDB:

```
const dbconnect = () => {
  let retries = 5;
  let retryInterval = 5000;
  const connectWithRetry = () => {
    mongoose
      .connect(URI, {
        useNewUrlParser: true,
        useUnifiedTopology: true,
        connectTimeoutMS: 20000,
        serverSelectionTimeoutMS: 5000,
        socketTimeoutMS: 45000,
        retryWrites: true,
      })
      .then(() => {
        console.log("DB is connected");
      })
      .catch((err) => {
        console.log("Failed connection: ", err);
        if (retries === 0) {
          console.log("Max retries reached. Exiting...");
          process.exit(1);
        } else {
          retries -= 1;
          console.log(`Reconnecting... Attempts left: ${retries}`);
          setTimeout(connectWithRetry, retryInterval);
        }
      });
  };

  connectWithRetry();
};
```

Si la conexión falla, el código intentará reconectar hasta 5 veces, con un intervalo de 5 segundos entre cada intento. Si después de 5 intentos no se ha podido conectar, el proceso se detendrá.

Opciones de Configuración:

- useNewUrlParser: habilita el nuevo analizador de URL de MongoDB para evitar posibles advertencias relacionadas con el analizador antiguo.
- useUnifiedTopology: Activa el nuevo motor de administración de conexiones, lo que mejora la estabilidad y el rendimiento de la conexión.

Tiempos de Espera:

- connectTimeoutMS: Define el tiempo máximo en milisegundos que el sistema esperará para establecer la conexión inicial con MongoDB.

- serverSelectionTimeoutMS: Establece el tiempo máximo que el cliente esperará para encontrar un servidor adecuado en el clúster de MongoDB.
- socketTimeoutMS: Configura el tiempo máximo que el cliente esperará antes de considerar que la comunicación con el servidor ha fallado.
- retryWrites: true: Esta opción permite que MongoDB reintente automáticamente las operaciones de escritura que fallan debido a ciertos errores de red o problemas transitorios, mejorando la resiliencia de la aplicación.

HERRAMIENTAS DE MONITOREO

Para monitorear la infraestructura, se utilizaron las siguientes herramientas: Prometheus, Grafana y MongoDB Exporter.

➢ **Prometheus**

Prometheus es una herramienta de monitoreo que recopila y almacena métricas de los servicios de la infraestructura. En este proyecto, se configuró Prometheus para recolectar métricas de MongoDB y del backend de la aplicación.

En el archivo docker-compose.yml, se configura Prometheus de la siguiente manera:

```
prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus:/etc/prometheus
    command:
      - "--config.file=/etc/prometheus/prometheus.yml"
    restart: always
    networks:
      - mongo-cluster
```

Este bloque indica que Prometheus utilizará la imagen oficial prom/prometheus y escuchará en el puerto 9090. Además, se configura para usar el archivo prometheus.yml, que define qué métricas recolectar.

El archivo prometheus.yml está configurado para monitorear los siguientes servicios:

- MongoDB Exporter: Para recolectar métricas de MongoDB.
- Prometheus: Para monitorear su propio estado.
- Backend: Para obtener métricas del backend de la aplicación

Archivo prometheus.yml:

```
global:
  scrape_interval: 10s

scrape_configs:
```

```
  - job_name: "mongodb"
    static_configs:
      - targets: ["mongodb_exporter:9216"]

  - job_name: "prometheus"
    static_configs:
      - targets: ["prometheus:9090"]

  - job_name: "backend"
    static_configs:
      - targets: ["backend:3000"]
```

El bloque *scrape_configs* define los servicios que Prometheus debe monitorear y las URL de destino desde donde recolectará las métricas.

Backend: Métricas Personalizadas

En el archivo *app.js* del backend, se agregó un middleware para contar las solicitudes HTTP entrantes y exponer métricas personalizadas a Prometheus:

```
const client = require('prom-client');
// Configuracion de metricas
const register = new client.Registry();
//metricas predeterminadas
client.collectDefaultMetrics({ register });
//Metricas personalizadas: Número de solicitudes HTTP
const httpRequestCounter = new client.Counter({
  name: "http_requests_total",
  help: "Número total de solicitudes HTTP",
  labelNames: ["method", "route", "status"]
});
register.registerMetric(httpRequestCounter);

// Middleware para rastrear solicitudes HTTP
app.use((req, res, next) => {
  res.on("finish", () => {
    httpRequestCounter.inc({
      method: req.method,
      route: req.route ? req.route.path : req.path,
      status: res.statusCode
    });
  });
  next();
});
// Endpoint de métricas
app.get('/metrics', async (req, res) => {
  res.set('Content-Type', register.contentType);
  res.end(await register.metrics());
});
```

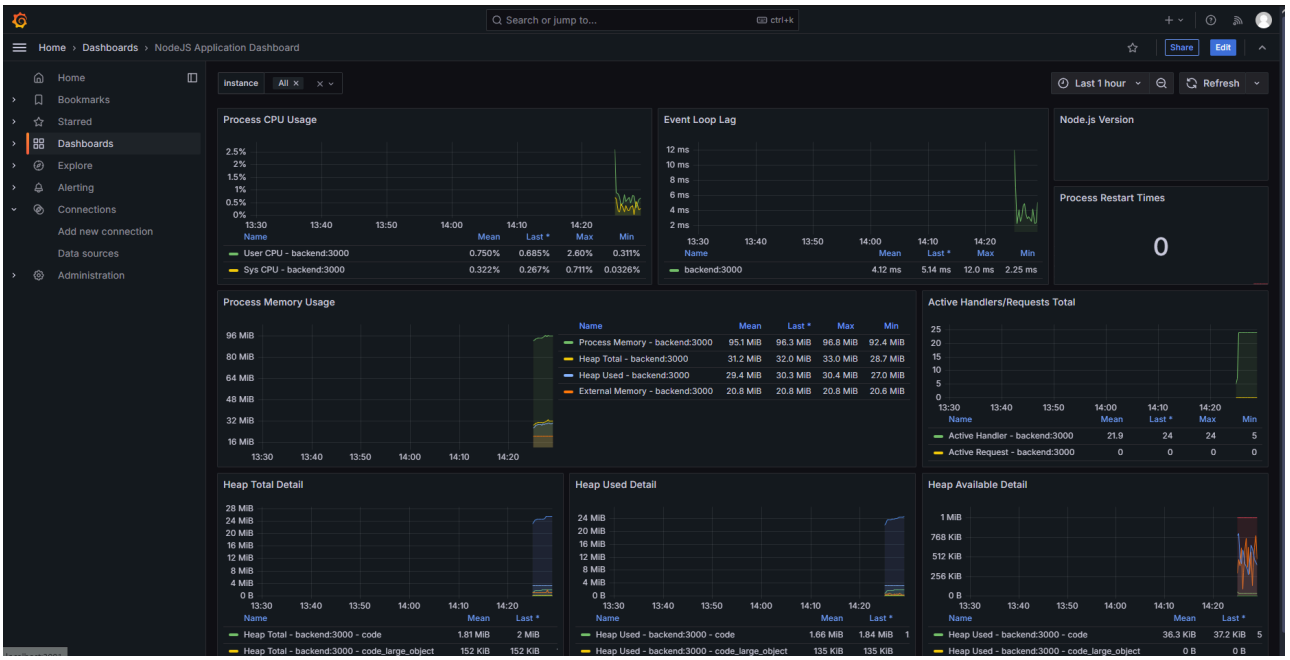
Este código usa **prom-client**, una biblioteca para Prometheus en Node.js, para exponer un endpoint /metrics que Prometheus puede consultar y recolectar métricas sobre las solicitudes HTTP.

➢ Grafana

Grafana se utiliza para visualizar las métricas recopiladas por Prometheus. En el archivo *docker-compose.yml:*

```
grafana:
    image: grafana/grafana:latest
    container_name: grafana
    ports:
      - "3001:3000"
    environment:
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=admin1234
    depends_on:
      - prometheus
    networks:
      - mongo-cluster
```

Grafana escucha en el puerto 3001 de la máquina local y se configura con un usuario y contraseña. Y declaramos que el contenedor depende de Prometheus para obtener las métricas a visualizar.



➢ MongoDBExporter

El MongoDB Exporter es una herramienta que expone métricas específicas de MongoDB en un formato que Prometheus puede entender. En el archivo docker-compose.yml:

```
mongodb-exporter:
    container_name: mongodb_exporter
    build:
      context: ./mongodb_exporter
      dockerfile: Dockerfile
    environment:
      - MONGODB_URI=mongodb://admin:admin1234@mongo_db:27017/
    ports:
      - "9216:9216"
    depends_on:
      - mongo_db
    entrypoint: ["dockerize", "-wait", "tcp://mongo_db:27017", "-timeout", "60s",
"/opt/bitnami/mongodb-exporter/bin/mongodb_exporter"]
    networks:
      - mongo-cluster
```

El mongodb exporter intenta por única vez conectarse a la base de datos (mongo_db) apenas se inicializan ambos, y dado que mongo_db esta montandose, desarrollando el cluster y cargando los datos, esta conexión es fallida. Es por esto que necesitamos que mongodb_exporter esperé que la base de datos esté lista para recién intentar la conexión, y para eso tenemos dockeriaze.

GitHub - jwilder/dockerize: Utility to simplify running applications in docker containers

Esta es una herramienta que nos permite esperar que otros servicios (contenedores) estén listos, realizando una comprobación a través de un protocolo específico (tcp, http) , para así iniciar el proceso principal (función principal del contenedor)

El Dockerfile:

```
FROM bitnami/mongodb-exporter:latest

USER root

# Descarga y descomprime el archivo
ADD
https://github.com/jwilder/dockerize/releases/download/v0.6.1/dockerize-linux
-amd64-v0.6.1.tar.gz /tmp/
RUN chmod +r /tmp/dockerize-linux-amd64-v0.6.1.tar.gz && \
    tar -xzvf /tmp/dockerize-linux-amd64-v0.6.1.tar.gz -C /tmp && \
    mv /tmp/dockerize /usr/local/bin/ && \
    chmod +x /usr/local/bin/dockerize
```

La imagen estará basada en bitnami/mongodb-exporter, dado que necesitaremos ingresar a carpetas protegidas del contenedor requerimos del superusuario.

Descargamos dockeriaze desde su repositorio oficial de github, descomprimimos y lo transferimos a la carpeta de archivos binarios (ejecutables) bin.

En la configuración del Docker-Compose establecemos como punto de partida para el proceso de ejecución del mongodb-exporter (entrypoint) con el siguiente comando:

```
entrypoint: ["dockerize", "-wait", "tcp://mongo_db:27017", "-timeout", "60s",
"/opt/bitnami/mongodb-exporter/bin/mongodb_exporter"]
```

Utiliza **dockerize** para:

Esperar (-wait) a que el servicio de MongoDB esté disponible en tcp://mongo_db:27017.

Establecer un tiempo límite (-timeout) de 60 segundos para esa espera.

Una vez que MongoDB esté listo, ejecuta el binario mongodb_exporter ubicado en /opt/bitnami/mongodb-exporter/bin/.

4. **Actividad 3**: Despliegue continuo

Objetivo: *Automatizar el flujo desde el repositorio hasta la producción:*

- **Configura un pipeline de integración y despliegue continuo (CI/CD):**
- **Herramientas recomendadas: GitHub Actions, GitLab CI/CD o Jenkins.**
- **Define un workflow que detecte cambios en el repositorio (p.ej., un push) y reconstruya la imagen del contenedor.**
- **Modifica el header/footer de la aplicación para incluir los nombres de los integrantes.**
- **Pruebas: Realiza un commit con los cambios y verifica que el despliegue sea automático**

DESPLIEGUE LOCAL

1- Crear en Github un workflow con el nombre docker-publish.yml con el siguiente contenido:

```
name: Build and Deploy Docker Image

on:
  push:
    branches:
      - master # Reemplaza con la rama principal de tu repositorio

jobs:
  build-and-push:
    runs-on: ubuntu-latest

    steps:
      # Paso 1: Clonar el repositorio
      - name: Checkout repository
        uses: actions/checkout@v3

      # Paso 2: Loguearte en Docker Hub
      - name: Log in to DockerHub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      # Paso 3: Construir la imagen Docker
      - name: Build Docker Image
        run: |
          docker build -t maxmartinez29/docker-resolve-frontend:latest -f
Frontend/Dockerfile ./Frontend

      # Paso 4: Subir la imagen a Docker Hub
      - name: Push Docker Image to Docker Hub
        run: |
          docker push maxmartinez29/docker-resolve-frontend:latest
```

Cuando pusheemos algún cambio al frontend, activará este workflow, generando una nueva imagen dentro de mi repositorio en Docker Hub.
https://hub.docker.com/repository/docker/maxmartinez29/docker-resolve-frontend/general

Dentro del docker-compose.yml, necesitamos un servicio con la imagen de watchtower, la cual nos ayuda a identificar cada 30 segundos si existe algún cambio en las imágenes de nuestros contenedores.

```
watchtower:
    image: containrrr/watchtower
    container_name: watchtower
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    command: --interval 30
    restart: always
```

El contenedor watchtower debería de actualizar la imagen de nuestro frontend y reiniciar el contenedor. De esta manera, tendremos nuestro contenedor de frontend con la imagen más reciente.

DESPLIEGUE CONTINUO ONLINE

El despliegue online permite que nuestra aplicación esté accesible desde cualquier lugar.

Para el proyecto decidimos usar **Render** para subir la parte del Backend y **Netlify** para el Frontend. Ambas plataformas aseguran que la aplicación se despliegue de manera continua y automática con cada cambio en el código,

Despliegue del Backend en Render

1. **Configuración del servicio**

Conectamos la cuenta de Raven con la de GitHub.

Seleccionamos crear un WebService y seleccionamos el repositorio donde tenemos el código del backend.

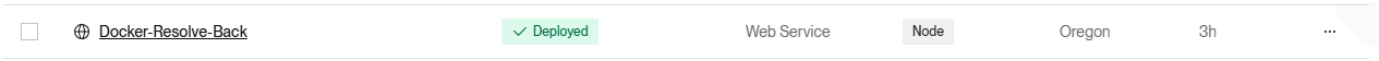
https://github.com/mariselAlarcon/Docker-Resolve-Deploy.git

Una vez seleccionado el repositorio, Render pide configurar algunas opciones:

- Nombre del servicio: Docker-Resolve-Back
- Branch: Elegimos la rama de Git 'master' que deseamos desplegar.
- Build Command: npm install
- Start Command: npm start
- Pre-Deploy Command: Backend , aquí se ingresa la dirección donde se encuentra nuestro archivo package.json, en nuestro caso en una carpeta denominada "Backend"
- Variables de entorno:
    - MONGO_URI: Decidimos subir nuestra base de datos en mongoDBAtlas entonces aqui se coloca la direccion que te proporciona dicha pagina.

Render detectará automáticamente el entorno de ejecución. Y en cuanto a las dependencias, Render instalará automáticamente las dependencias de tu aplicación según el archivo package.json si está configurado para Node.js.

2. **Despliegue automático**



Una vez terminado el proceso Render proporciona una URL pública para acceder al backend.

https://docker-resolve-back.onrender.com

Render detectará automáticamente cualquier cambio en la rama 'master' del repositorio. Cuando se realice un push a esa rama, Render desplegará los cambios de manera automática.

Despliegue del Frontend en Netlify

1. **Cuenta en Netlify**

Creamos una cuenta en la plataforma y la asociamos con la cuenta de GitHub donde tenemos nuestro repositorio.

2. **Configuración del Sitio**

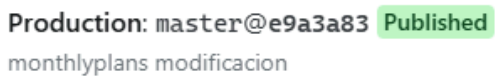
Buscamos la opción Add New Site y seleccionamos Exporting existing Project para seleccionar el repositorio de GitHub propio.

Después de conectar el repositorio, Netlify te pedirá configurar algunos detalles:

- Branch: elegimos la rama de Git que deseamos desplegar, master.
- Build Command: ng build --configuration production
- Publish Directory: Frontend/dist/resolve-gym-website/browser
  Este es el directorio donde se encuentran los archivos estáticos que se deben publicar.

Una vez configurado todo, hacemos clic en "Deploy site"

3. **Despliegue automático**



Una vez terminado el proceso Netlify proporciona una URL pública para acceder al sitio.

https://resolvedeploy.netlify.app

Al igual que Render, Netlify detecta automáticamente los cambios que realices en tu repositorio. Cada vez que subas cambios a la rama seleccionada, Netlify realizará un nuevo despliegue de tu aplicación frontend.

Una vez que ambos servicios están configurados, se establece un flujo de **Integración y Despliegue Continuo (CI/CD)** Esto asegura que nuestra aplicación esté siempre actualizada en producción sin necesidad de intervención manual.

Modificaciones en el código del Frontend:

Para una correcta gestión de la URL del backend, creamos una carpeta **environments**, dentro de la cual creamos dos archivos environments una para desarrollo y otro que entra en un entorno de ejecución.

Archivo environments.prod.ts

```
export const environment = {
```

```
    production: true,
    apiUrl: 'https://docker-resolve-back.onrender.com'};
```

Archivo environments.ts

```
export const environment = {
    production: false, // En desarrollo
    apiUrl: 'http://localhost:3000' // URL de tu backend en desarrollo
  };
```

Dentro de cada servicio de nuestro frontend, modificamos la URL que redirecciona al backend ahora ubicado en la plataforma de Render.

```
import { environment } from '../../environments/environment';
...
 private baseUrl = `${environment.apiUrl}/monthlyPlans`;
```

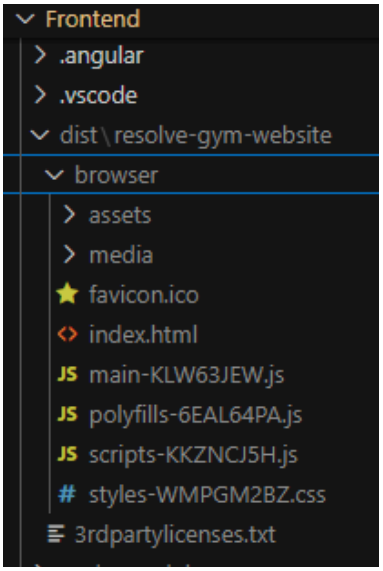
¿Qué ocurre durante el despliegue en Netlify?

Cuando Netlify realiza el despliegue del frontend, ejecuta el comando de construcción especificado en la configuración previa:

```
ng build --configuration production
```

Este comando tiene dos efectos principales:

- Reemplazo del archivo de configuración de entornos: El archivo environment.ts se reemplaza por environment.prod.ts, lo que actualiza las configuraciones del entorno, como la URL del backend, la cual apunta al backend desplegado en Render.
- Generación de los archivos estáticos: El comando crea la carpeta de producción, que contiene todos los archivos estáticos necesarios para que Netlify pueda servir la aplicación correctamente.



También debemos modificar el código del Backend para que permita entradas de peticiones desde el Netlify.

En el archivo app.js:

```
app.use(cors({
  origin: ['http://localhost:4200', 'https://resolvedeploy.netlify.app']
}));
```

5. **Actividad 4:** Clúster de replicación

Objetivo: *Configurar un clúster de replicación para la base de datos.*

---

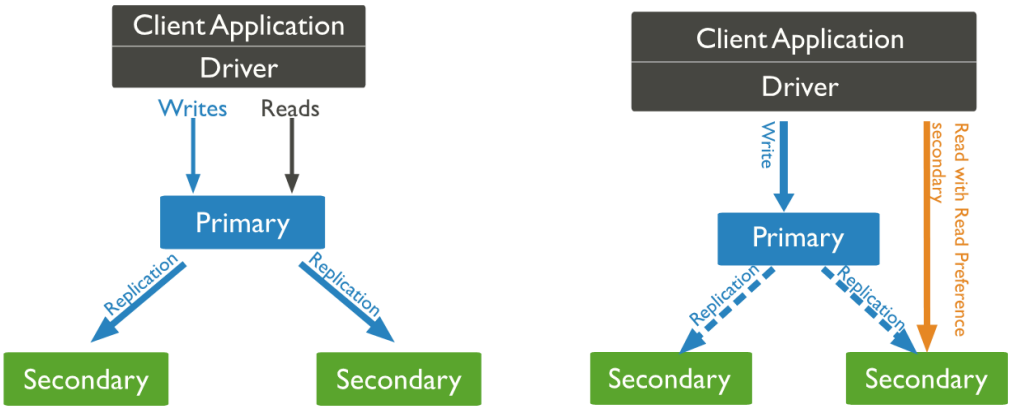
1. **Implementa un clúster de replicación para la base de datos (p. ej., PostgreSQL). Herramientas: Patroni, Pgpool-II, o servicios internos de MySQL/MongoDB.**

Replica Sets de MongoDB

Un **Replica Set** es un grupo de instancias de MongoDB que mantienen las mismas copias de datos. En este modelo, **los datos se replican automáticamente entre los nodos**, proporcionando tolerancia a failovers y alta disponibilidad.

Los **Replica Set** de MongoDB ofrecen una replicación maestro esclavo, en dónde **únicamente el nodo primario puede recibir consultas de escritura y lectura**, mientras que los nodos secundarios solo pueden recibir consultas de lectura.

Además, es posible configurar la conexión a una base de datos MongoDB para que tenga como preferencia de lectura a nodos secundarios, **dirigiendo las operaciones de lectura a estos en lugar del primario para aligerar su carga**.



Cómo funciona la replicación en un Replica Set

1. **El nodo primario registra las operaciones en el oplog**
    - Cada operación de escritura (inserción, actualización o eliminación) realizada en el nodo primario se guarda en el oplog.
    - El oplog es un registro secuencial que almacena las operaciones realizadas en la base de datos, pero no los resultados de las mismas. Por ejemplo:
        - Una operación de inserción se registra como: `insert document X en la colección Y`.
        - Una operación de eliminación se registra como: `delete document X de la colección Y`.

- Esta **forma de replicación es de tipo asincrónica**, ya que el nodo primario responde al cliente tan pronto como registra la operación en su log, sin esperar confirmaciones de los secundarios.

Característica	Replicación Síncrona	Replicación Asincrónica
Confirmación de operación	Se espera a que todos (o algunos) nodos secundarios confirmen la operación antes de responder al cliente.	El nodo primario responde al cliente tan pronto como registra la operación en su log, sin esperar confirmaciones de los secundarios.
Latencia	Mayor, debido a que las escrituras deben propagarse y confirmarse antes de continuar.	Menor, ya que no se espera a que los secundarios confirmen.
Consistencia	Más fuerte, ya que los datos están garantizados de estar sincronizados en todos los nodos involucrados.	Eventual, ya que los secundarios pueden tener un pequeño retraso al replicar las operaciones.
Disponibilidad	Puede verse afectada si los secundarios están lentos o no disponibles, porque las escrituras no pueden continuar.	Más resiliente, ya que las operaciones no dependen directamente de los secundarios.

2. **Los nodos secundarios replican el oplog**
   - Los nodos secundarios se conectan al primario y comienzan a leer las entradas del **oplog** en orden cronológico.
   - Cada nodo aplica esas operaciones localmente para mantener sus datos sincronizados con el primario.
3. **Sincronización inicial**
   - Cuando se agrega un nuevo nodo a un Replica Set, realiza una sincronización inicial copiando todos los datos existentes desde el primario o un secundario designado.
   - Después de la sincronización inicial, comienza a replicar las nuevas operaciones del oplog en tiempo real.
4. **Mantenimiento del estado del Replica Set**
   - Cada nodo monitorea continuamente el estado de los demás mediante mensajes de "latido" (*heartbeats*).
   - Si un nodo secundario se queda desactualizado (por ejemplo, debido a un fallo de red), utiliza el oplog para ponerse al día.

Manual de Replication Sets en MongoDB.

Instrucciones para configurar un Replica Set en Docker

La configuración de un Replica Set en MongoDB implica dos pasos principales: **inicializar las instancias** y **configurar el Replica Set**.

1. Para que un Replica Set funcione, es necesario **iniciar cada nodo que lo compone como una instancia de MongoDB configurada específicamente para el ReplicaSet**. Esto se realiza utilizando el comando `mongod` en cada nodo, con las siguientes opciones:

```
mongod --replSet mongoReplicaSet --bind_ip localhost,mongo_db
```

   - `--replSet`: Especifica el nombre del Replica Set (en este caso, `mongoReplicaSet`).
   - `--bind_ip`: Define las interfaces desde las cuales el nodo acepta conexiones. En este ejemplo, se permite la conexión local (`localhost`) y desde un contenedor llamado `mongo_db`.

2. Una vez inicializado todos los nodos, debemos **inicializar el Replica Set con** `rs.initiate`. Para ello vamos a entrar al cliente de línea `mongosh` y ejecutar el siguiente comando:

```
rs.initiate({
  _id: "mongoReplicaSet", // Nombre del replica set (debe coincidir con
--replSet)
  members: [
    { id: 0, host: "mongo_db:27017" }, // Nodo primario
    { _id: 1, host: "mongo_db1:27017" }, // Nodo secundario
    { _id: 2, host: "mongo_db2:27017" }  // Nodo secundario (opcional)
  ]
});
```

Después de ejecutar `rs.initiate`, puedes **verificar que el Replica Set se haya configurado correctamente utilizando el comando `rs.status()`**. Este comando **muestra información sobre el estado del Replica Set, los nodos conectados y su rol (primario o secundario).**

Para completar el primer paso, vamos a indicarle a nuestro archivo `docker-compose.yml` que deseamos correr el comando de inicialización de instancia al iniciar cada nodo con `command`.

```
mongo_db:
    ...
    command: mongod --replSet mongoReplicaSet --bind_ip localhost,mongo_db
```

Desplegar un Cluster de MongoDB con Docker.

Adición de KeyFiles

Algo que se debe tener en consideración, al configurar las variables de entorno nuestro nodo principal `mongo_db`, MongoDB nos solicitará **inicializar las instancias con un keyfile compartido**. Un keyfile es una **clave compartida utilizada para autenticar la comunicación entre los miembros del conjunto de réplicas**. Este archivo debe ser idéntico en todas las instancias del conjunto.

Además, este archivo debe contar con los **permisos mínimos de lectura**, y que el propietario sea el **usuario de MongoDB (999)** para garantizar la seguridad de este archivo.

Para ello, vamos a sobrescribir el `entrypoint` predeterminado de MongoDB para ejecutar un script personalizado que se ejecutará al iniciar el contenedor, de esta forma vamos a cambiar los permisos del archivo antes de usarlo con `command`.

```
mongo_db:
    ...
    volumes:
    ...
      - ./mongo_key/replica.key:/data/replica.key.devel
    ...
    entrypoint:
      - bash
      - -c
      - |
          cp /data/replica.key.devel /data/replica.key
          chmod 400 /data/replica.key
          chown 999:999 /data/replica.key
          exec docker-entrypoint.sh $$@
    command: mongod --replSet mongoReplicaSet --bind_ip localhost,mongo_db
--keyFile /data/replica.key
```

Es importante no olvidar ejecutar el entrypoint de docker al cambiar los permisos del `replica.key`, de lo contrario no se inicializará correctamente el contenedor de docker.

También cabe mencionar que al momento de copiar nuestra `replica.key` dentro de cada nodo, es necesario cambiarle el nombre para que al cambiar los archivos no se modifique nuestro archivo local y afecte nuestras operaciones.

Update Self-Managed Replica Set to Keyfile Authentication - MongoDB Manual v8.0.

Set up keyfile for replica set in docker-compose · Issue #475 · docker-library/mongo · GitHub.

Script de Inicialización de Replica Set

Una vez configurada la inicialización de cada nodo, sigue configurar el Replica Set como tal. Para ello, modificamos el script de inicialización `init-mongo.sh` para incluir este comando,

```
...
```

```
mongo=( mongosh --username admin --password admin1234 --quiet )

echo '[init-mongo.sh] Inicializando Replica Set...'
"${mongo[@]}" <<-EOF
  rs.initiate({
        _id: 'mongoReplicaSet',
        members: [
        {_id: 0, host: 'mongo_db', priority: 2},
        {_id: 1, host: 'mongo_db1', priority: 1},
        {_id: 2, host: 'mongo_db2', priority: 1}
        ]
    });
EOF

...
```

También configuramos la prioridad de los nodos, que los ayudarán a seleccionar un nodo principal en caso de que sea necesario un cambio (como el failover del principal o cambios en la configuración del Replica Set).

Además, añadimos estructuras de control que verifican que el contenedor se encuentre operativo antes de realizar cada comando:

Estructura de Control para mongod

```
: "${FORKED:=}"

if [ -z "${FORKED}" ]; then
  echo >&2 '[init-mongo.sh] mongod para la inicialización va a apagarse'
  mongod --pidfilepath /tmp/docker-entrypoint-temp-mongod.pid --shutdown
  echo >&2 '[init-mongo.sh] el conjunto de réplicas se inicializará más tarde'

  FORKED=1 "${BASH_SOURCE[0]}" &
  unset FORKED
  mongodHackedArgs=(:)
  return
fi

echo '[init-mongo.sh] Proceso de MongoDB init completado, asegurando que MongoDB
este listo para la inicialización del conjunto de replicas.'
```

Este control detiene temporalmente MongoDB para realizar configuraciones previas, reejecuta el script en un subproceso para continuar con tareas de inicialización (el `command` con la inicialización de la instancia del replica set) sin bloquear el proceso principal y prepara a MongoDB para el proceso de inicialización del Replica Set.

[ugly hack to initialize replica set for MongoDB docker container, put under /docker-entrypoint-initdb.d/ · GitHub](#).

Estructura de Control de Conexión a BD

```
mongo=( mongosh --username admin --password admin1234 --quiet )

tries=30
while true; do
  sleep 1
  if "${mongo[@]}" --eval 'quit(0)' &> /dev/null; then
    # Exito
    break
  fi
  (( tries-- ))
  if [ "$tries" -le 0 ]; then
    echo >&2
    echo >&2 '[init-mongo.sh] error: no se pudo iniciar Replica Set'
    echo >&2
    kill -STOP 1 # initdb no se ejecurá una segunda vez.
    exit 1
  fi
done
```

Este control verifica que se haya inicializado correctamente MongoDB y que está listo para aceptar conexiones. Si está disponible, el flujo continúa normalmente.

Estructura de Control para esperar la Configuración del Replica Set

```
cho '[init-mongo.sh] Esperando a que el Replica Set esté listo...'

# Esperar hasta que haya un nodo primario disponible en el Replica Set
until "${mongo[@]}" --eval "rs.status()" | grep 'PRIMARY'; do
  echo "[init-mongo.sh] Esperando a que el nodo primario esté disponible..."
  sleep 5
done

echo '[init-mongo.sh] Replica Set inicializado exitosamente. Procediendo con la
restauración de la base de datos...'
```

Este control se ejecuta luego de configurar el Replica Set, que esperará que se complete la configuración del mismo antes de proceder con la restauración de la base de datos.

2. **Cambia la configuración de tu aplicación para que apunte al clúster en lugar de a la instancia aislada.**

Configuración de la URI en el Back-end

El back-end usa una string de conexión llamada URI para conectarse con nuestra base de datos Mongo. Ahora, para que en lugar de que apunte a una sola instancia, la modificaremos para que apunte al Replica Set. Para ello, debemos configurar la URI para incluir todos los nodos del Replica Set, especificar el nombre del replica set, y, opcionalmente, ajustar algunos parámetros relacionados con la conexión.

```
const URI = process.env.MONGO_URI ||
'mongodb://admin:admin1234@mongo_db:27017,mongo_db1:27017,mongo_db2:27017/gym_db?
authSource=admin&replicaSet=mongoReplicaSet&readPreference=secondaryPreferred';
```

Connection String Examples - MongoDB Manual v8.0.

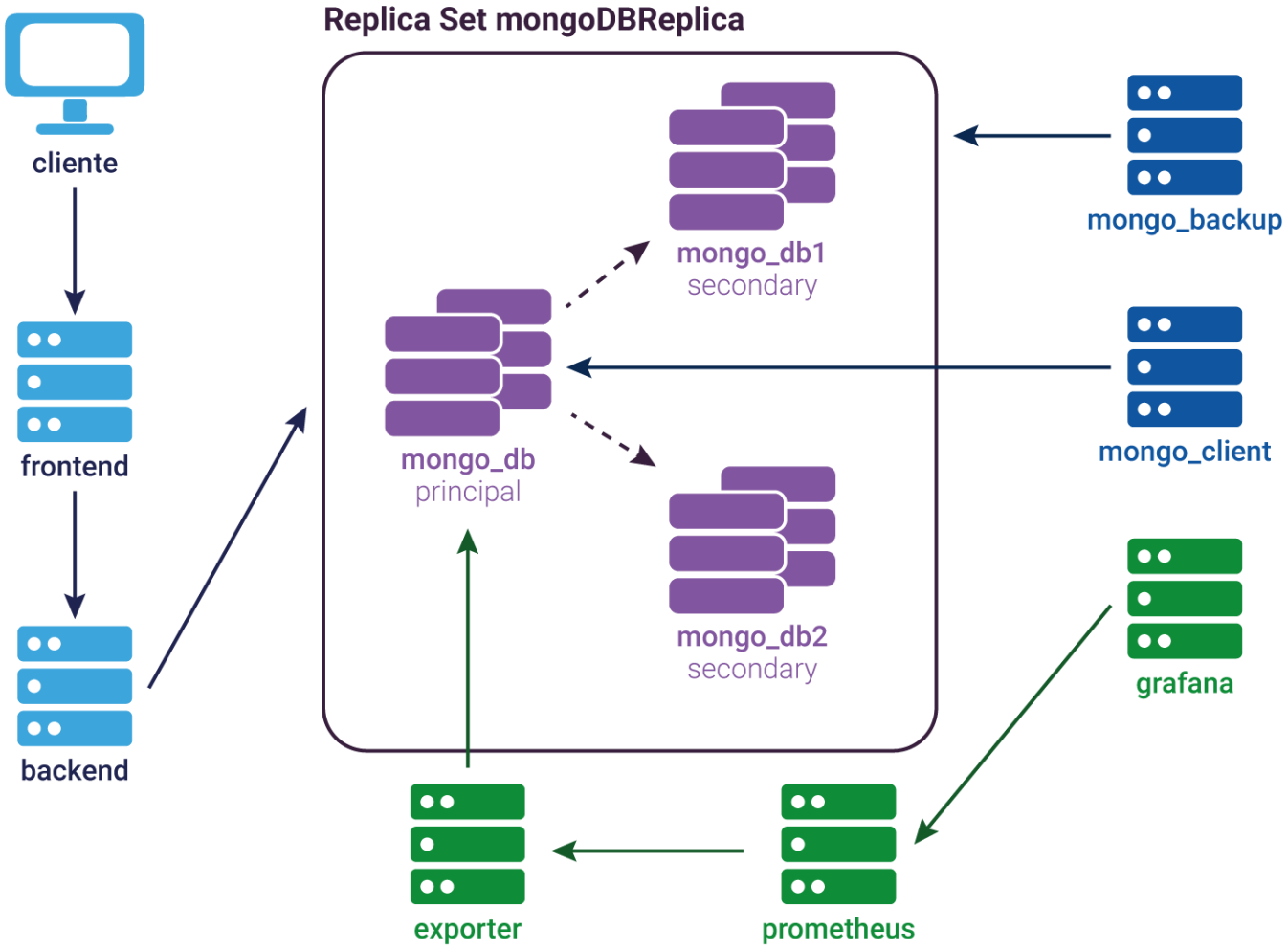
Configuración del Back-up

De forma similar, podemos configurar el comando de `mongodump` de nuestro contenedor mongo-backup para que se conecte con nuestra Replica Set especificando el nombre de nuestro conjunto y los nodos participantes:

```
mongodump
--host="mongoReplicaSet/mongo_db:27017,mongo_db1:27017,mongo_db2:27017"
```

Diagrama de Arquitectura

Arquitectura Dockerizada



Arquitectura Desplegada