Launcher Dockerizado NEST MS

- Vamos a crear contenedores para cada microservicio
- Cada uno de estos directorios sea una imagen de Docker, y que con hacer docker run y el nombre del microservicio se eche a correr
- Vamos a tenr que crear las imágenes basadas en una arquitectura de procesador según dónde vayan a correr
- De múltiples arquitecturas, para amd64, x86, etc
- Con CI/CD podremos construir la imagen en la nube, no será mi pc quién la construya
- Se desplegará automáticamente a través de Kubernetes (hay que habilitarlo desde DockerDesktop, reiniciar y darle a Reset Kubernetes cluster)
- En producción no voy a querer correr nest start:dev
- De hecho no necesito a nest para hacer el build, lo puedo hacer con node

---

Construir a producción

- En producción no voy a querer correr nest start:dev
- De hecho no necesito a nest para hacer el build, lo puedo hacer con node
- Puedo hacer un build con **npm run build** del client-gateway y correr con **node dist/main.js**
- Dará error porque no se está comunicando con nadie a través de NATS, pero devuelve un error 500 al llamar a un endpoint desde POSTMAN
- Para crear la imagen de Docker para producción de client-gateway, crearemos Dockerfile.prod
- Para reducir el tamaño de la imagen de Docker y eviatr usar el usuario root para la construcción de la imagen
- Haremos el multi-stage build
- Dividiremos el dockerfile en 3 etapas: dependencias, builder, crear la imagen final
- Para identificar que viene una nueva etapa se hace una refrencia a una nueva imagen, en este caso node:21, la llamo build
- Me muevo al working directory
- Se recomienda que los comandos que tienden menos a cambiar se coloquen al front, ya que pueden mantenerse en caché
- Copio de deps (así llamé a la primera imagen donde instalé las dependencias) las dependencias que hay en node_modules al ./node_modules del working directory
- La ruta es la misma que indiqué en la imagen anterior
- Puedo limpiar los módulos de node que no necesito y dejar solo los de producción. Esto reducirá el peso de mi imagen a la mitad
- Agrego una nueva etapa que llamo prod para la imagen final
- Siempre indico el working directory, el directorio donde estoy trabajando, donde trabaja la imagen de node:21
- Copio de la imagen anterior llamada build los node_modules
- Copio el codigo fuente de la carpeta dist a ./dist (la carpeta dist del working directory) que ya tiene el código de JS
- Le indico que la variable NODE_ENV será de production

- Hago uso del usuario node porque el usuario que viene por defecto en la imagen de node:21 tiene demasiados privilegios
- Expongo el puerto 3000. Los contenedores son pequeñas computadoras virtuales, no habrá ningún conflicto de puertos
- Ejecuto el comando de node dist/main.js

```
# Dependencias
FROM node:21-alpine3.19 as deps

WORKDIR /usr/src/app

COPY package.json ./
COPY package-lock.json ./

RUN npm install


# Builder - Construye la aplicación
FROM node:21-alpine3.19 as build

WORKDIR /usr/src/app

# Copiar de deps, los módulos de node
COPY --from=deps /usr/src/app/node_modules ./node_modules

# Copiar todo el codigo fuente de la aplicación
COPY . .

# RUN npm run test
RUN npm run build

# para limpiar los módulos de node que no necesito y dejar solo los de producción
(opcional)
# reducirá el peso de la imagen
RUN npm ci -f --only=production && npm cache clean --force



# Crear la imagen final de Docker
FROM node:21-alpine3.19 as prod

WORKDIR /usr/src/app


COPY --from=build /usr/src/app/node_modules ./node_modules

# Copiar la carpeta de DIST
COPY --from=build /usr/src/app/dist ./dist


ENV NODE_ENV=production
```

```
USER node


EXPOSE 3000

# El comando a ejecutar para correr la imagen
CMD [ "node", "dist/main.js" ]
```

- Uso docker build -f nombredelarchivo -t (de tag) el nombre de la aplicación y punto para que busque el archivo en el contexto dónde está

```
docker build -f dockerfile.prod -t client-gateway .
```

- En containers le doy al play y en optional settings introduzco el nombre client-gateway-prod
- En ports coloco 3000 : 3000/tcp
- En environment variables coloco **variable=PORT value=3000 y NATS_SERVERS value=nats://nats:4222**

Launcher

- Quiero que mediante un comando pueda construirlas todas
- Es casi idéntico que el docker-compose.yml pero más simple, ya que no hay que exponer ciertos puertos ni ciertos comandos porque las imágenes ya estarán construidas
- No ocupamos los puertos del nats-server porque eran puertos de monitoreo
- Necesitamos que client-gateway apunte a mi Dockerfile.prod
- En build le indico el contexto y el dockerfile
- Le indico el nombre de la imagen con image
- El puerto y los volúmenes no son necesarios en producción. El puerto es necesario solo para ejecutar.
- El volumen es para tener un enlace entre mi src y el src del contenedor, para que cuando haya un cambio se recargue todo. Tampoco lo necesito
- El command no es necesario ni las variables de entorno tampoco, ya está especificado en el dockerfile
- Si usara el docker compose con up si necesitaría el puerto y las variables de entorno
- Indicamos el resto de imagenes en el docker-compose
- Para orders necesito el aprovisionamiento de una base de datos postgres en la nube, porque no voy a usar ni levantar una imagen de postgres dede aqui
- Usaremos Serverless Postgres, con la version de postgres 16 que me dará una cadena de conexión
- El de payments se conecta a través del puerto ya que es híbrido y se comunica por http y por NATS
- En orders uso args para mandarle un argumento en el momento de la construcción que es la variable de entorno de la db postgres en la nube
-

```
version: '3'


services:

  nats-server:
```

```
    image: nats:latest

  client-gateway:
    build:
      context: ./client-gateway
      dockerfile: dockerfile.prod
    image: client-gateway-prod
    ports:
      - ${CLIENT_GATEWAY_PORT}:${CLIENT_GATEWAY_PORT}
    environment:
      - PORT=${CLIENT_GATEWAY_PORT}
      - NATS_SERVERS=nats://nats-server:4222


  auth-ms:
    build:
      context: ./auth-ms
      dockerfile: dockerfile.prod
    image: auth-ms
    environment:
      - PORT=3000
      - NATS_SERVERS=nats://nats-server:4222
      - DATABASE_URL=${AUTH_DATABASE_URL}
      - JWT_SECRET=${JWT_SECRET}

  products-ms:
    build:
      context: ./products-ms
      dockerfile: dockerfile.prod
    image: products-ms
    environment:
      - PORT=3000
      - NATS_SERVERS=nats://nats-server:4222
      - DATABASE_URL=file:./dev.db


  # Orders MS
  orders-ms:
    build:
      context: ./orders-ms
      dockerfile: dockerfile.prod
      # uso args para mandarle la variable de entorno con el string de conexión
      args:
        - ORDERS_DATABASE_URL=${ORDERS_DATABASE_URL}
    image: orders-ms
    environment:
      - PORT=3000
      - DATABASE_URL=${ORDERS_DATABASE_URL}
      - NATS_SERVERS=nats://nats-server:4222


# ======================
# Payments Microservice
# ======================
```

```
  payments-ms:
    build:
      context: ./payments-ms
      dockerfile: dockerfile.prod
    image: payments-ms
    ports:
      - ${PAYMENTS_MS_PORT}:${PAYMENTS_MS_PORT}
    environment:
      - PORT=${PAYMENTS_MS_PORT}
      - NATS_SERVERS=nats://nats-server:4222
      - STRIPE_SECRET=${STRIPE_SECRET}
      - STRIPE_SUCCESS_URL=${STRIPE_SUCCESS_URL}
      - STRIPE_CANCEL_URL=${STRIPE_CANCEL_URL}
      - STRIPE_ENDPOINT_SECRET=${STRIPE_ENDPOINT_SECRET}
```

- El Dockerfile-prod de auth, es prácticamente el mismo que el de client-gateway
- Solo que tengo que usar prisma generate, genera el cliente y lo almacena en los módulos de node
- Se podrían aplicar migraciones en este punto, siempre y cuando la db esté definida

```
# Dependencias
FROM node:21-alpine3.19 as deps

WORKDIR /usr/src/app

COPY package.json ./
COPY package-lock.json ./

RUN npm install



# Builder - Construye la aplicación
FROM node:21-alpine3.19 as build

WORKDIR /usr/src/app

# Copiar de deps, los módulos de node
COPY --from=deps /usr/src/app/node_modules ./node_modules

# Copiar todo el codigo fuente de la aplicación
COPY . .

# RUN npm run test
RUN npm run build

RUN npm ci -f --only=production && npm cache clean --force

RUN npx prisma generate
```

```
# Crear la imagen final de Docker
FROM node:21-alpine3.19 as prod

WORKDIR /usr/src/app


COPY --from=build /usr/src/app/node_modules ./node_modules

# Copiar la carpeta de DIST
COPY --from=build /usr/src/app/dist ./dist


ENV NODE_ENV=production

USER node


EXPOSE 3000

CMD [ "node", "dist/main.js" ]
```

- En el Dockerfile.prod de orders

```
# Dependencias
FROM node:21-alpine3.19 as deps

WORKDIR /usr/src/app

COPY package.json ./
COPY package-lock.json ./

RUN npm install




# Builder - Construye la aplicación
FROM node:21-alpine3.19 as build

# necesito pasarle la variable de entorno de la db postgres en la nube con args,
la recojerá del docker-compose.prod.yml
ARG ORDERS_DATABASE_URL
# le digo que la DATABASE_URL, que es la variable de entorno que uso en
environment del docker-compose.prod.yml es esta variable de args
ENV DATABASE_URL=$ORDERS_DATABASE_URL
# todo esto es necesario para que prisma recoja la variable de entorno para que
cuando llegue a la migración tenga la url y se pueda ejecutar

WORKDIR /usr/src/app

# Copiar de deps, los módulos de node
```

```
COPY --from=deps /usr/src/app/node_modules ./node_modules

# Copiar todo el codigo fuente de la aplicación
COPY . .


# aqui se puede hacer la migración, para este punto la db debería de existir
RUN npx prisma migrate deploy
RUN npx prisma generate


# RUN npm run test
RUN npm run build

RUN npm ci -f --only=production && npm cache clean --force




# Crear la imagen final de Docker
FROM node:21-alpine3.19 as prod

WORKDIR /usr/src/app


COPY --from=build /usr/src/app/node_modules ./node_modules

# Copiar la carpeta de DIST
COPY --from=build /usr/src/app/dist ./dist


ENV NODE_ENV=production

USER node


EXPOSE 3000

CMD [ "node", "dist/main.js" ]
```

- En el Dockerfile de products

```
# Dependencias
FROM node:21-alpine3.19 as deps

WORKDIR /usr/src/app

COPY package.json ./
COPY package-lock.json ./

RUN npm install
```

```
# Builder - Construye la aplicación
FROM node:21-alpine3.19 as build

WORKDIR /usr/src/app

# Copiar de deps, los módulos de node
COPY --from=deps /usr/src/app/node_modules ./node_modules

# Copiar todo el codigo fuente de la aplicación
COPY . .

# RUN npm run test
RUN npm run build

RUN npm ci -f --only=production && npm cache clean --force

# si la variable de entorno no está definida este comando no se ejecutará
# uso args en el docker-compose.prod.yml
RUN npx prisma generate



# Crear la imagen final de Docker
FROM node:21-alpine3.19 as prod

WORKDIR /usr/src/app


COPY --from=build /usr/src/app/node_modules ./node_modules

# Copiar la carpeta de DIST
COPY --from=build /usr/src/app/dist ./dist
# Hay que clonar la carpeta de prisma desde build
COPY --from=build /usr/src/app/prisma ./prisma


ENV NODE_ENV=production

USER node


EXPOSE 3000

CMD [ "node", "dist/main.js" ]
```

> docker compose -f docker-compose.prod.yml build
