UCC- Universidad Católica de Córdoba

Facultad de Ingeniería

INGENIERÍA DE SOFTWARE IV

Trabajo Práctico II

Integrantes:

- Cuozzo, Sofia
- Hernandez, Simon

Docente:

- Schwindt, Ariel
- Bono, Fernando

INFORME TRABAJO PRÁCTICO 2 - DOCKER

1. Introducción

El objetivo del trabajo fue contenerizar una aplicación Web API en .NET 7 (C#) y configurarla para correr en entornos QA y Producción con bases de datos independientes en MySQL 8.0.

Se buscó aplicar buenas prácticas de DevOps, utilizando Dockerfiles multi-stage, separación de variables de entorno y persistencia de datos en volúmenes. Además, se diseñó un esquema de versionado de imágenes y pruebas de conectividad entre la aplicación y la base de datos.

2. Tareas a realizar

De acuerdo a la consigna, las tareas principales fueron:

1. Elección de la aplicación y framework
    - Implementar Web API en .NET 7.
    - Definir endpoints básicos para pruebas y operaciones sobre la base de datos.

2. Contenerización con Docker
    - Crear Dockerfile multi-stage con imágenes oficiales de Microsoft.
    - Configurar build y runtime en etapas separadas.

3. Base de datos
    - Definir servicios MySQL para QA y PROD con volúmenes persistentes.
    - Configurar inicialización automática mediante variables de entorno.

4. Entornos QA y PROD
    - Configurar docker-compose.yml con servicios separados y variables de entorno distintas.
    - Garantizar independencia de datos entre QA y PROD.

5. Versionado y publicación de imágenes
    - Publicar imágenes en Docker Hub con tags latest y v1.0.
6. Pruebas de funcionamiento
    - Verificar endpoints de ping y seed en QA y PROD.
    - Comprobar persistencia tras reinicio de contenedores.

3. ¿Cómo las realizamos?

1. Aplicación Web API
    - Se desarrolló en .NET 7 con endpoints /WeatherForecast, /db/ping, /db/init, /db/seed y /db/all.
    - Se utilizó MySqlConnector para integración con la base.

2. Dockerfile multi-stage

- Build stage: mcr.microsoft.com/dotnet/sdk:7.0 → compila y publica en modo Release.
- Runtime stage: mcr.microsoft.com/dotnet/aspnet:7.0 → ejecuta en contenedor liviano y seguro.
- Se expuso el puerto 80 y se configuró la ejecución con dotnet SimpleWebAPI.dll.

```
#  Build y publish

FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build

WORKDIR /src



# Copiamos solo el csproj para cachear el restore

COPY SimpleWebAPI/SimpleWebAPI.csproj SimpleWebAPI/

RUN dotnet restore SimpleWebAPI/SimpleWebAPI.csproj



# Ahora sí copiamos el resto del código

COPY SimpleWebAPI/ SimpleWebAPI/



# Compilamos y publicamos

WORKDIR /src/SimpleWebAPI

RUN dotnet publish -c Release -o /app/publish /p:UseAppHost=false



#  Runtime

FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS final

WORKDIR /app

COPY --from=build /app/publish .

EXPOSE 80

ENTRYPOINT ["dotnet", "SimpleWebAPI.dll"]
```

3. **Configuración de entornos**

   - Se definieron dos servicios para la app (app-qa y app-prod), ambos basados en la misma imagen (soficuozzo/simplewebapi:v1.0).
   - Cada servicio se conectó a su propia base MySQL (db-qa y db-prod) en redes y volúmenes distintos.

Docker-compose.yml

```
networks:
  qa_net:
  prod_net:


volumes:
  mysqldata_qa:
  mysqldata_prod:


services:
  # ---------- QA ----------
  db-qa:
    image: mysql:8.0
    container_name: mysql-simplewebapi-qa
    environment:
      MYSQL_ROOT_PASSWORD: "SofiCuoZZo#2025!"
      MYSQL_DATABASE: "simpledb_qa"         # se crea sola al
iniciar
    command:
--default-authentication-plugin=mysql_native_password
    volumes:
      - mysqldata_qa:/var/lib/mysql
```

```
    networks:
      - qa_net
    restart: unless-stopped
    healthcheck:
      test: ["CMD-SHELL", "mysqladmin ping -h localhost
-p$${MYSQL_ROOT_PASSWORD} || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 12


  app-qa:
    image: soficuozzo/simplewebapi:v1.0
    container_name: simplewebapi-qa
    environment:
      ASPNETCORE_ENVIRONMENT: "QA"
      # Connection string para MySQL:
      # - Server=db-qa (nombre del servicio)
      # - AllowPublicKeyRetrieval=True evita handshake fallido
con MySQL 8
      # - SslMode=None simplifica en local
      ConnectionStrings__Default:
"Server=db-qa;Port=3306;Database=simpledb_qa;User
ID=root;Password=SofiCuoZZo#2025!;AllowPublicKeyRetrieval=True;Ss
lMode=None;"
    depends_on:
      db-qa:
        condition: service_healthy
```

```
    ports:
      - "8080:80"
    networks:
      - qa_net
    restart: unless-stopped


  # ---------- PROD ----------
  db-prod:
    image: mysql:8.0
    container_name: mysql-simplewebapi-prod
    environment:
      MYSQL_ROOT_PASSWORD: "SofiCuoZZo#2025!"
      MYSQL_DATABASE: "simpledb_prod"
    command:
--default-authentication-plugin=mysql_native_password
    volumes:
      - mysqldata_prod:/var/lib/mysql
    networks:
      - prod_net
    restart: unless-stopped
    healthcheck:
      test: ["CMD-SHELL", "mysqladmin ping -h localhost
-p$${MYSQL_ROOT_PASSWORD} || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 12
```

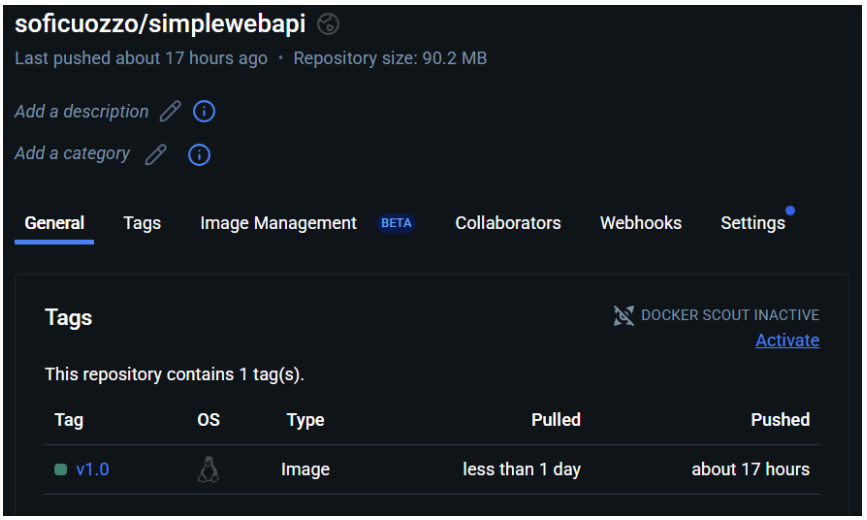
```
  app-prod:
    image: soficuozzo/simplewebapi:v1.0
    container_name: simplewebapi-prod
    environment:
      ASPNETCORE_ENVIRONMENT: "Production"
      ConnectionStrings__Default:
"Server=db-prod;Port=3306;Database=simpledb_prod;User
ID=root;Password=SofiCuoZZo#2025!;AllowPublicKeyRetrieval=True;Ss
lMode=None;"
    depends_on:
      db-prod:
        condition: service_healthy
    ports:
      - "8081:80"
    networks:
      - prod_net
    restart: unless-stopped
```

4. **Persistencia de datos**
    - Se crearon volúmenes dedicados: mysqldata_qa y mysqldata_prod.
    - Esto garantizó independencia total de la información entre QA y PROD.

5. **Versionado y publicación**
    - Imágenes subidas a Docker Hub bajo el usuario soficuozzo.
    -

docker build -t uccsimon/simplewebapi:v1.0

docker push uccsimon/simplewebapi:v1.0




```
PS C:\Users\sofic\Documents\FACULTAD 2025\SEGUNDO SEMESTRE\INGENIERIA DE SOFTWARE 3\TP2-NCuozzo-Hernande-Inge 3> docker push soficuozzo/simplewebapi:v1.0
The push refers to repository [docker.io/soficuozzo/simplewebapi]
6cba3d36311b: Pushed
5ec18103e025: Layer already exists
270f7fde987a: Layer already exists
4d29f6e29d10: Layer already exists
b4ec6db9c251: Layer already exists
ba941484fbe1: Layer already exists
123eef91533f: Layer already exists
v1.0: digest: sha256:3dfe0ec60e28f8b5608991dd22cb29baae84a606f0116e3813af8f20790de151 size: 1788
```

6. Pruebas

    - Con curl se validó /db/ping en ambos entornos (200 OK).
    - Se probó que los datos insertados en QA no aparezcan en PROD.
    - Tras reiniciar contenedores, los datos se mantuvieron gracias a los volúmenes.

curl.exe -i http://localhost:8080/db/ping

curl.exe -i -X POST http://localhost:8080/db/init

curl.exe -i -X POST http://localhost:8080/db/seed

curl.exe -i http://localhost:8080/db/all

4. Problemas encontrados y soluciones

- Error 500 en /db/ping al inicio
    - La aplicación se levantaba antes que MySQL.
    - Solución: se agregaron healthcheck y depends_on: condition: service_healthy.
- Problema de autenticación con MySQL 8
    - El conector no aceptaba el plugin por defecto.
    - Solución: configurar --default-authentication-plugin=mysql_native_password y parámetros en la cadena de conexión (AllowPublicKeyRetrieval=True;SslMode=None).
- Datos mezclados entre QA y PROD
    - Al inicio se usaba un volumen compartido.
    - Solución: separar en mysqldata_qa y mysqldata_prod, además de redes distintas (qa_net y prod_net).

5. Conclusión

El trabajo permitió contenerizar una aplicación en .NET 7 siguiendo buenas prácticas de DevOps:

- Se implementaron entornos QA y Producción totalmente aislados.
- Se garantizó persistencia de datos con volúmenes.
- Se resolvieron problemas típicos de sincronización entre aplicación y base.

- Se aplicó versionado claro en Docker Hub, facilitando releases estables.